软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

数值稳定性相关安全漏洞隐患的自动化检测方法 (软件安全漏洞检测专刊)*

沈维军[1,2], 汤恩义[1,2*], 陈振宇[1,2], 陈 鑫[1,3], 李 彬[1,3], 翟 娟[1,2]

[1](南京大学 计算机软件新技术国家重点实验室,江苏 南京 210023)

[2](南京大学 软件学院,江苏 南京 210093)

[3](南京大学 计算机科学与技术系,江苏 南京 210023)

通讯作者: 汤恩义, E-mail: eytang@nju.edu.cn

摘 要: 安全漏洞检测是保障软件安全性的重要手段.随着互联网的发展,黑客的攻击手段日趋多样化,且攻击技术不断翻新,使软件安全受到了新的威胁.本文描述了当前软件中实际存在的一种新类型的安全漏洞隐患,我们称之为数值稳定性相关的安全漏洞隐患.由于黑客可以利用该类漏洞绕过现有的防护措施,且已有的数值稳定性分析方法很难检测到该类漏洞的存在,因而这一新类型的漏洞隐患十分危险.面对这一挑战,本文首先从数值稳定性引起软件行为改变的角度定义了数值稳定性相关的安全漏洞隐患,并给出了对应的自动化检测方法.该方法基于动静态相结合的程序分析与符号执行技术,通过数值变量符号式提取、静态攻击流程分析、以及高精度动态攻击验证三个步骤,来检测和分析软件中可能存在的数值稳定性相关安全漏洞.我们在业界多个著名开源软件上进行了实例研究,实验结果表明,本文方法能够有效检测到实际软件中真实存在的数值稳定性相关漏洞隐患.

关键词: 漏洞检测;数值稳定性;程序分析;软件安全
中图法分类号: TP311

中文引用格式: 沈维军,汤恩义,陈振宇,陈鑫,李彬,翟娟.数值稳定性相关安全漏洞隐患的自动化检测方法.软件学报. http://www.jos.org.cn/1000-9825/0000.htm

英文引用格式: Shen WJ, Tang EY, Chen ZY, Chen X, Li B, Zhai J. An Approach of Vulnerability Detection related to Numerical Stability. Ruan Jian Xue Bao/Journal of Software, 2017 (in Chinese). http://www.jos.org.cn/1000-9825/ 0000.htm

Automated Detection of Vulnerability related to Numerical Stability

SHEN Wei-Jun[1,2], TANG En-Yi[1,2], CHEN Zhen-Yu[1,2], CHEN Xin[1,3], LI Bin[1,3], ZHAI Juan[1,2]

[1](State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China)

[2](Software Institute, Nanjing University, Nanjing 210093, China)

[3](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

+ Corresponding author:TANG En-Yi. E-mail:eytang@nju.edu.cn

Abstract: Vulnerability detection is an important way of improving the security of software. However, the development of the Internet makes it possible for hackers to attack software systems with new techniques. So the software is still vulnerable. This paper describes a new kind of vulnerability. We call it the vulnerability related to numerical stability. As hackers are able to bypass the security protection by the new kind of vulnerability, and numerical analysis is difficult to detect such a vulnerability, it is dangerous. In this paper, we define the vulnerability related to numerical stability from the perspective of software behavior variation caused by numerical errors, and further

* 基金项目: 国家自然科学基金(00000000, 00000000); 南京大学计算机软件新技术国家重点实验室开放课题(KFKT00000000)

Foundation item: National Natural Science Foundation of China (00000000, 00000000); State Key Laboratory for Novel Software Technology (Nanjing University)开放课题 (KFKT00000000)

收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00

CNKI 在线出版时间: 0000-00-00

propose an automatic detection approach. The approach combines the static analysis and symbolic execution, and detects the vulnerability by three steps: symbolic extraction, static attack analysis, and dynamic attack verification with high precision values. We evaluate our approach on a few famous open source projects. The results show that our approach effectively detects the vulnerabilities related to numerical stability hidden in the real-world projects.

Key words: vulnerability detection; numerical stability; program analysis; software security

1 引言

随着互联网的高速发展与移动应用网络的日益普及，黑客的攻击频率逐年增多，软件安全成为了当前学术界与工业界共同关注的重要议题。据国际著名软件安全供应商赛门铁克(Symantec)公司统计[1,2]，2016 年被检测到的网络攻击总数比 2011 年增长了 48 倍。不仅如此，黑客的攻击手段复杂多样，攻击技术不断翻新，各种新类型的恶意软件不断涌现，使计算机软件安全面临着新的挑战。在如何有效检测现有软件中可能存在的安全漏洞隐患，并在第一时间内进行修复防范，以减轻或避免因软件安全问题而造成损失等问题上，学者们进行了大量研究，取得了重要进展[3-5]。然而，在软件中存在的数值稳定性相关安全漏洞隐患目前尚未得到足够的重视。

软件数值稳定性是指软件数值输出在各变量有微小误差时仍能保持基本不变的能力。许多计算机软件采用浮点数格式进行数值计算，不可避免地会引入计算误差。随着软件规模和复杂性的增加，数值误差会逐渐累积，造成程序的计算结果偏离真实值，从而使软件发生数值错误。长期以来，数值分析领域的专家学者对这一问题进行了深入研究，取得了丰硕的成果[6-12]，使数值计算结果的可靠性大大增加。然而这些研究主要关注于数值误差的累积量，对于误差累积较少即可引入的软件安全漏洞却无能为力。近年来，业界出现了一些利用数值稳定性相关漏洞的安全攻击[13,14]，由于这些攻击利用了人们容易忽视的数值稳定性漏洞，很容易绕过传统的软件安全技术防护措施，因而使得软件安全面临严重威胁。因此，业界需要针对这类新安全漏洞的检测与防范措施。

图 1 给出了一个数值稳定性相关安全漏洞的程序片段示例。该程序循环在每一轮迭代时接受一个浮点类型的用户输入 input，并将它累计到求和变量 sum 中去。当输入值较大，满足 input>1 时，指针 p 指向一个 A 类型的对象，而当输入值特别小，满足 input<1E-6 时，求和变量 sum 与上一轮循环迭代的求和变量值 sum_old 之差也会小于 1E-6，这时指针 p 会指向一个 B 类型的对象。该程序的开发人员认为 input>1 和 sum- sum_old<1E-6 不会同时得到满足，因此在循环迭代结束时对应地释放了 p 所指向的对象。然而，黑客可以利用代码的数值稳定性构造一个简单的攻击输入流，使程序出现设计开发人员意料之外的软件行为，从而威胁软件的安全性。在图 1 所示的程序片段中，黑客仅需要依次输入 2 个 input：1.00E09 和 1.01，该程序就会因数值稳定性问题而造成 input>1 和 sum-sum_old<1E-6 同时得到满足，导致代码存在 double free[15,16]的安全风险[1]。如果该软件的设计开发人员在安全性验证环节忽略了软件的数值误差积累与数值稳定性的相关论证，他们甚至会错误地证明没有控制流在这段代码中会引发 double free。正因为如此，这类在代码中真实存在的安全漏洞隐患可以逃过现有的漏洞检测技术而变得十分危险。值得注意的是，即使该软件的设计开发人员考虑了数值稳定性问题，他们所关注的一般也仅仅是误差累积的多少，在该示例程序中的漏洞被利用时，其对应的绝对误差值为 1.01，相对误差值为 1.01E-9。这样的误差值累积在数值计算中一般并不会引起注意，因而传统的数值稳定性分析方法也很难检测到这些数值稳定性相关的安全漏洞隐患。我们需要新的安全漏洞检测技术来发现这类问题。

根据已知的数值稳定性相关安全漏洞现状推断，当数值误差的累积会引起程序的软件行为发生改变时，对应的软件会存在数值稳定性相关的安全漏洞隐患。此时，数值误差的累积量并不一定很大，软件输出也并

[1] 当采用浮点数对数量级差别较大的数值进行运算时,会因为数值稳定性而产生问题.这里的黑客攻击使得 sum 发生了 1.00E09+1.01 的运算,理论上计算结果应该会大于 1.00E09,但由于浮点数精度的限制,计算机实际的运行结果是等于 1.00E09 的,因此在这样的输入下 sum=sum_old.黑客就是利用这一特点来构造攻击,威胁软件的安全性.

不一定偏离真实值很远。软件行为的改变主要体现在以下两个方面：1.软件的控制流发生了改变；2.软件的数据依赖发生了变化。当数值误差的积累引起软件的控制流发生变化时，软件可能会执行到开发人员意料之外的程序路径，从而使得软件存在被黑客截获敏感信息、篡改关键数据，或者中断正常的软件执行过程等的危险，图 1 的示例程序即因为数值误差积累而引起了软件的控制流发生改变。另外，当数值误差的累积引起软件的数据依赖发生了变化，则会使软件的污点特征(taint characteristic)发生改变，从而使得黑客能够扩大其访问权限的范围。因此，当数值误差积累引发上述类型的软件行为发生改变时，我们认为软件安全会在一定程度上受到威胁。

```
1  float sum, sum_old, input = 0.1f;
2  while (input > 0.0f)
3  {   scanf("%f", &input);
4      sum += input;
5      if (sum - sum_old < 1E-6f)
6      {   p = new A;
7      }
8      if (input > 1.0f)
9      {   p = new B;
10     }
11     ......
12     if (sum - sum_old < 1E-6f)
13     {   output(p.a);
14         delete p;
15     }
16     if (input > 1.0f)
17     {   output(p.b);
18         delete p;      //存在double free风险
19     }
20     sum_old = sum;
21 }
```

Fig.1 Risk of Double Free Caused by the Vulnerability Related to Numerical Stability

图 1 数值稳定性相关的安全漏洞导致代码存在 double free[15,16]的安全风险

本文基于动静态相结合的程序分析与符号执行技术来自动检测数值稳定性相关的软件安全漏洞隐患。通过数值变量符号式提取、静态攻击流程分析等步骤，我们分别从由误差引起控制流变化和数据依赖变化的角度分析可能的攻击输入与攻击流程。并通过将原始待测软件转换成高精度计算，并动态比对高精度计算和原始精度计算下的关键执行路径与内存数据，来验证安全漏洞隐患是否真实存在。从而实现数值稳定性相关安全漏洞的自动化检测。

我们在业界多个著名开源软件上进行了实例研究，实验结果表明，数值稳定性相关的安全漏洞隐患与实际软件中数值误差的积累程度无关，很多误差积累很小的模块也同样会存在安全漏洞隐患。正因为如此，数值稳定性相关的安全漏洞隐患在许多含有数值计算代码的实际软件中普遍存在，且安全漏洞隐患的数量同软件中数值计算的代码量正相关。另外，实验结果也说明了本文方法能够有效检测到实际大型软件中真实存在的数值稳定性相关安全漏洞隐患。

本文后续部分将按如下方式组织：第二节我们定义并解释了数值稳定性相关安全漏洞隐患的概念，并对误差的引入方式进行了详细介绍；第三节具体描述了数值稳定性相关安全漏洞隐患的检测方法；在第四节中，我们描述了在 9 个著名开源项目上的实验过程，并给出了实验结果；第五节介绍了本文的相关工作；最终第六节总结全文，并得出结论。

2 数值稳定性相关的漏洞隐患

由于很好的平衡了效率与可用性，IEEE 754 所定义的浮点数算术标准在工业界得到了广泛应用。许多现有软件均通过单精度类型或者双精度类型的浮点数来进行数值计算。图 2 列出了这两种浮点数格式的表示形式，一般来说每个浮点数值由 1 位符号位、w 位指数位、以及 f 位有效数字位构成。当符号位的取值为 s，指数位的取值为 E，有效数字位的取值为 M 时，浮点格式所表示的数值为：

$$(-1)^S * M * 2^{E-f+1}$$

然而，由于字长的限制这一表示形式不可避免地会引入误差，这些误差是由数值的四舍五入产生的，一般我们也称之为舍入误差(roundoff error)。在实际浮点数运算中，舍入误差的引入形式一般有两种，当程序直接用浮点数来表示一个实数值时，由于浮点数值表示能力的限制而引入的误差，我们称之为表示误差。例如我们直接将实数值 0.2 赋值给一个浮点变量时，浮点变量中保存的值就会有表示误差。而在数值计算中由浮点数的操作而引入的误差我们称为操作误差，例如我们将 1.00E-200 的值加到 9.00E200 上去，而在结果中产生的误差即为操作误差。

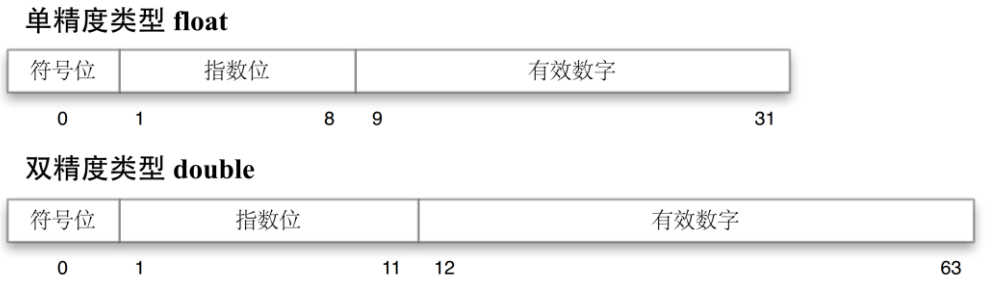

Fig.2 IEEE 754 Floating-point Formats

图 2 IEEE 754 定义的浮点数格式

通过增加浮点数的表示内存，可以有效减少误差的引入。基于这一原理，高精度计算和任意精度计算近年来被引入到数值计算中[17]，使误差问题得到有效解决。然而，由于任意精度计算不仅需要占用大量的存储空间，其计算速度也会比直接使用 IEEE 754 所定义的浮点数算术标准慢近千倍。这在很多应用领域是无法接受的，因此，很多软件仍然以 IEEE 754 浮点数标准进行开发。为了解决误差问题，数值分析与软件工程领域的学者提出了自动化数值稳定性分析来估计误差的积累程度[10,18-20]。

然而，已有的研究均未关注到数值稳定性相关的安全漏洞问题。近年来，由于网络安全问题日益突出，数值稳定性相关的安全漏洞隐患也凸显出来：一方面，误差的累积可能会导致软件的执行代码发生变化，从而引发安全性问题。黑客可以蓄意构造攻击输入使误差累积到一定程度，从而造成软件按照黑客的目的执行，引起安全性问题。引言中描述的图 1 的例子就因数值稳定性问题导致软件的控制流发生变化，从而存在 double free 的漏洞隐患。

另外，数值稳定性也会造成软件的数据依赖发生变化，从而导致对应软件存在安全漏洞隐患。由于数据依赖变化的实质是黑客访问权限的变化，因此，数值误差造成数据依赖变化的漏洞隐患对于软件会产生重大威胁。例如，某程序以浮点值的计算结果来决定分配内存的大小，黑客通过误差积累使内存的分配量小于实

际的使用量，从而构造出缓冲区溢出等攻击方案。在这一过程中，软件的数值稳定性问题并未使得程序执行不同的代码，软件的控制流也未曾发生变化，但内存数据的依赖性却发生了变化。软件的数值稳定性使得原来在缓冲区外黑客不能访问到的内存变得能够被黑客访问到了。因此，数值稳定性相关的数据依赖也是软件存在安全漏洞隐患的重要因素之一。

因此在本文中，当软件因数值稳定性问题而引起软件行为发生变化时，我们就称该软件存在数值稳定性相关的安全漏洞隐患。而软件行为的变化在本文中特指控制流的变化，和数据依赖的变化。本文是基于这一思想来对数值稳定性相关的软件安全漏洞隐患进行检测的。

3 漏洞隐患检测方法

本文基于动静态相结合的程序分析与符号执行技术来检测软件中存在的数值稳定性相关安全漏洞隐患。图 3 给出了本文漏洞隐患检测方法的整体框架。基于这一框架，我们通过三个检测阶段来最终获得待分析软件中存在的安全漏洞隐患。首先，第一阶段通过符号执行将数值代码中各浮点变量的符号式从其计算逻辑中提取出来，用于后续的攻击流程分析；然后，第二阶段通过静态分析方法搜索可能会对程序行为变化造成影响的攻击输入；由于静态分析可能存在误报(false positive)，该框架的第三阶段将使用高精度计算来动态验证攻击输入对应的攻击效果，并最终确认可能存在的安全漏洞隐患。下面我们将分别对这三个阶段的具体技术作详细说明。

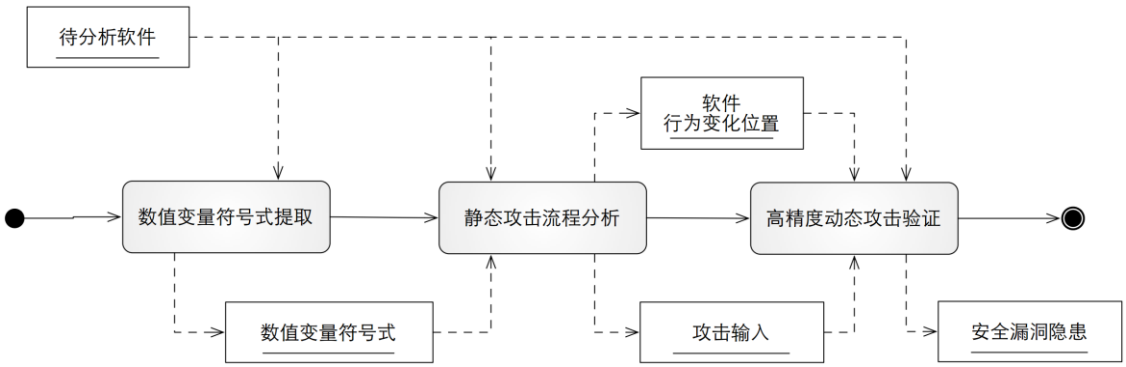

Fig.3 The Detection Framework of Vulnerabilities Related to Numerical Stability

图 3 数值稳定性相关漏洞隐患的检测方法整体框架

3.1 数值变量符号式提取

作为数值稳定性相关漏洞隐患检测的第一步，数值变量符号式提取将待分析软件中各浮点变量的计算过程从软件中分离出来，以便后续步骤进一步进行误差分析与攻击流程分析。浮点变量的计算过程将通过一个累积符号表达式来表示，例如，在软件中有 float a=x*5；在经过一些程序逻辑后 float b=a+y；经过数值变量符号式提取后，b 的值机会被更新成 x*5+y。这样的数值变量符号表达式 e 在后续的攻击分析和误差分析时非常有用。

在数值变量符号式中除了需要记录符号表达式 e 之外，还需要记录一个当前的条件约束 c，以标识该表达式的执行条件。例如，在软件中如果有这样条件分支 **if**(x<5) b=x+5; **else** b=x*5; 经过这条语句后，b 的值就会被映射成一对二元组(c,e)，当 c 为 x<5 时，e 的值为 x+5，而当 c 为 x≥5 时，e 的值为 x*5。

我们通过符号执行(Symbolic Execution)来构建程序中每一个数值变量到符号式二元组的映射，由于后续算法中我们需要使用到数值变量在待测试软件中的中间值，我们对每一个数值变量在待测试软件中的更新点进行了标定，得到了数值变量的标号，例如，对于变量 a 在程序中的多个更新点，我们将其标识为 l_{a1},l_{a2},l_{a3},...

等等，然后我们将标号 l_{a1} 映射到符号式二元组即可。

算法 1 描述了数值变量符号式的提取过程，它通过在符号执行来完成条件约束的更新，并输出两个映射 *L* 和 *SYM*，其中映射 *L* 将软件中的各个数值变量映射到其更新位置的标号上，而映射 *SYM* 将各更新位置的标号映射到符号式的二元组。符号执行通过构建执行状态 ES 来记录程序各路径的条件约束，每一个执行状态 ES 内部由一个四元组(*pc,c,sv,cv*)构成，其中程序计数器 *pc* 记录了执行状态 ES 在程序中的当前位置，路径约束 *c* 记录了执行状态当前的路径约束，符号值集 *sv* 记录了软件到达当前位置时各符号变量的表达式，具体变量集 *cv* 记录了程序中非符号值，例如 int x=3，此时符号执行引擎会直接将 x 的内存地址和具体值 3 记录到执行状态 ES 中。

算法 1 符号式提取函数

```
输入：ExecState initState;
输出：L, SYM
初始化：Queue<ExecState> ESQueue;                    //执行状态队列
 1: ESQueue.enqueue(initState);
 2: while ESQueue.size>0 do
 3:    ES = ESQueue.dequeue();
 4:    if ES.pc == EXIT then                          //到达程序出口
 5:       delete ES; continue;
 6:    end if
 7:    if ES.pc == FORK then  //到达条件跳转 (包括循环/分支)
 8:       cond = condition(ES.pc);
 9:       ES.c = ES.c ∧cond;
10:       ES2 = new ExecState(ES);
11:       ES2.c = ES.c ∧ ¬cond;
12:       propagateStep(ES2);
13:       ESQueue.enqueue(ES2);
14:    end if
    //到达变量更新点 l_v, 需把数值变量 v 更新成 exp 的值
15:    if ES.pc == {v ← exp}^lv then
16:       e = updatesym(ES.sv,exp);
17:       if l_v ∈ L(v) then
18:          SYM(l_v)=SYM(l_v)∪{(ES.c,e)};
19:       else
20:          L(v).add(l_v);
21:          SYM(l_v)={(ES.c,e)};
22:       end if
23:    end if
24:    propagateStep(ES);
25:    ESQueue.enqueue(ES);
26: end while
```

符号执行引擎在软件入口处会构建一个初始的执行状态 initState，其程序计数器 *pc* 为程序入口，路径约束 *c* 为 True，符号值集 *sv* 为程序输入，且各输入的初值为可代表任意值的符号，具体变量集 *cv* 为空。算法 1 通过执行状态队列 ESQueue 沿着软件的各条路径不断推进执行状态的运行，且每次取一个执行状态，直至所有执行状态均到达程序出口。在默认情况下，符号执行根据执行状态 ES 的当前指令 pc 执行引擎的默认操作 propagationStep(ES)使程序计数器 ES.*pc* 向前前进一步(算法 1 的第 24 行)，并更新执行状态 ES 中的符号值集 *sv* 与具体变量集 *cv*。当符号执行引擎到达程序的条件跳转位置时(算法 1 的第 7 行)，我们首先从条件跳转中提取条件约束 cond，并分离出一个新的执行状态 ES2，它和原始状态分别指向条件跳转的两个分支方向，并记录不同的路径约束。当符号执行到达浮点变量 *v* 的更新点时 l_v时，算法 1 通过函数 updatesym 将当前状

态的符号值集 ES.*sv* 中各个符号值分别代入用于更新变量 *v* 的表达式 *exp*，从而获得数值变量符号表达式 *e*。并将当前标号更新到输出映射 *L* 中，当前路径约束 ES.*c* 和数值变量符号表达式 *e* 更新到输出映射 *SYM* 中。

3.2 静态攻击流程分析

静态攻击流程分析主要探讨软件的攻击输入，即通过静态分析来获得使数值误差的积累引发软件行为发生改变的程序输入。基于本文对软件行为发生改变的定义，我们分别从数值误差积累引起控制流变化以及误差积累引起软件数据依赖变化的角度分别探讨攻击流程的分析方法。

3.2.1 控制流变化分析

软件的控制流程由各分支条件以及循环条件决定。当某个条件的真假值发生变化时，软件执行的控制流即发生对应改变。例如 if *cond* then ... else ... endif 中当浮点误差使得本来应该取 true 值的条件约束 *cond* 取到了 false，则软件数值稳定性就引起了程序的控制流发生变化。因此，本文的控制流变化分析重点关注数值误差影响到的条件约束。我们将程序中条件约束 *cond* 的文法规范化定义如下：

cond → *relexp* | ¬ *relexp* | *cond lop relexp*

relexp → *arithexp relop 0.0*

lop → *AND* | *OR*

relop → < | > | = | ≥ | ≤ | ≠

这里的 *relexp* 表示一个经过了规范化的关系表达式，这些关系表达式通过二元逻辑操作符 *lop* 连接起来，或者通过一元逻辑操作符¬ 取反。在规范化关系表达式的内部，由关系运算符 *relop* 连接算术表达式和 0 构成。当实际程序中出现右部不为 0 的关系表达式时(例如 $f_1 < f_2$)，本文会将其规范化成右部为 0 的关系表达式(f_1-f_2 < 0.0)来进一步分析使条件约束发生真假值转换的前提。

对于待测软件中的每一个条件约束 *cond*，面向控制流变化的攻击流程分析首先扫描其逻辑操作符 *lop* 构建的逻辑关系，即得到条件约束中哪些关系表达式取值的变化会影响整体条件约束的取值变化。然后针对每一个关系表达式 *relexp* 讨论其取值发生变化的误差范围。例如，对于 f_1-f_2 < 0.0 && f_2-f_3 > 0.0 这样的条件约束，我们重点讨论怎样的浮点误差会使 f_1-f_2 以及 f_2-f_3 的精确值和浮点结果会产生不同的正负号。我们称这一状况为约束 f_1-f_2 < 0.0(或 f_2-f_3 > 0.0)在精确实数计算条件下和浮点数值计算条件下真值相反。

对于软件中的某一个数值 *f*，我们将不考虑误差时 *f* 的精确实数值记为 *REAL(f)*，而将程序经过浮点计算获得的浮点值记为 *FP(f)*，则我们有：

$$REAL(f) = FP(f) + round(f) \tag{1}$$

这里 *round(f)*为数值 *f* 在程序中的总体舍入误差。许多已有的数值分析，以及稳定性分析方法可以帮助我们判断舍入误差的范围[11,21,22]。特别地，当数值变量符号式提取算法已经获得了 *f* 的计算表达式时，本文方法通过仿射分析[22]来获得(affine analysis)对应的误差表达式 *round(f)*。

对于经过规范化的关系表达式 *relexp*，我们不失一般性地讨论其中的三种关系运算符>、< 和 =，而对于另外三种关系运算符≥、≤和≠可以通过逻辑操作符合我们讨论的三种关系运算符复合而获得。例如： $f \neq 0$ 可以写成¬ $f=0$ 的形式。下面我们讨论如何找到使精确实数计算条件下和浮点数值计算条件下真值相反的关系表达式。

定理 1：当且仅当

$$|round(f)| > |FP(f)| \tag{2}$$

且

$$FP(f) * round(f) \leq 0 \tag{3}$$

时，关系表达式 *f relop 0* 的真值结果会因为数值误差而发生变化，即 *FP(f) relop 0* 和 *REAL(f) relop 0* 的真值相反。

证明：先证明充分性。当$|round(f)| > |FP(f)|$和$FP(f) * round(f) < 0$成立时，首先$FP(f)$ *relop* 0,，必然

存在真值 T 或者 F，又因为$REAL(f) = FP(f) + round(f)$,，由条件*(2)(3)*可知，此时如果FP(f)为正值或负值，那么$REAL(f)$必然变号，所以真值就发生了翻转；如果此时$FP(f)$为 *0*，那么$REAL(f) = round(f) > 0$也必然不等于 0，此时的真值也发生了翻转。得证。

再证明必要性。当简单判断语句的真值结果发生实数域和浮点域上的翻转时，我们也分两种情况讨论，当$FP(f)$不等于 *0* 时，不妨设$FP(f)$为正值，此时因为$REAL(f) = FP(f) + round(f)$,，要使$REAL(f)$的值变成负值,，必然要使$|round(f)| > |FP(f)|$且$round(f)$为负值，所以(2)(3)成立。当$FP(f)$等于 0 时，此时要使$REAL(f)$的值不等于 0，必然要求$|round(f)|>0$，此时(2)(3)也成立，所以得证。 □

本文控制流分析的目的在于找到对应的攻击输入使精确实数计算条件下和浮点数值计算条件下关系表达式的真值相反。鉴于目前 *f* 和 *round(f)*的符号式已知，按照定理 1 的要求，我们通过构建约束来获得攻击输入。所构建的攻击约束 *attcond* 为：

$$|round(f)| - f > 0 \ \wedge \ round(f) * f \leq 0 \tag{4}$$

当攻击约束 *attcond* 在约束求解器中可解时，我们通过约束求解直接获得对应的攻击输入值，而当软件逻辑较为复杂，使攻击约束 *attcond* 中存在非线性分量时，我们通过随机搜索方式搜索攻击约束的可满足解，从而获得满足条件的攻击输入。

3.2.2 数据依赖变化分析

除了控制流程的改变，数据依赖的变化也同样会造成软件存在安全漏洞问题。由于数据依赖变化的实质是黑客访问权限的变化，例如，原先并不依赖于环境变量的密码值等敏感信息，由于数值稳定性问题而产生了依赖关联，并使得黑客有权限能够访问到这一部分信息。因此，数值误差造成数据依赖变化的漏洞隐患会对于软件产生重大威胁。

从理论上说，污点分析(Taint Analysis)方法[23,24]，被称为信息流追踪技术，是处理和分析数据依赖，防止数据完整性和保密性被破坏的有效手段。该技术通过对系统中敏感数据进行标记，继而跟踪标记数据在程序中的传播，以检测系统安全问题。。例如，当我们把软件输入等不被信任的数据源看作"被污染"的数据时，污点分析可以获得"被污染"数据的波及范围，即黑客可能能访问或修改的数据范围。然而传统的污点分析方法一般未考虑到数值误差对"被污染"数据的影响，即由于数值误差的积累可能引起"被污染"数据的波及范围发生变化，即本文所述的数值稳定性相关数据依赖变化。因此面向数据依赖变化攻击流程分析可以采用引入误差计算的污点分析和不引入误差计算的污点分析获得各自的"被污染"数据范围，再通过比较这两个数据范围的差异来分析攻击输入。然而，经过我们的实践证明，这样的分析方式效率太低。结合精度分析的污点分析需要消耗大量的计算资源，因此本文在实现上采用了针对当前计算机体系结构的优化分析方案，来解决检测由数值误差引起的数据依赖变化及其相关的漏洞隐患。

当前主流的计算机体系结构中，一般用整数值来标定内存地址。因此现代程序设计语言的申请内存、分配内存、释放内存等等存储操作一般也都用整数值来标定其大小和偏移。整数类型是程序中数据传播不可或缺的重要标识。我们发现，软件的浮点数值误差一般都要通过影响整数类型值从而进一步影响到数据依赖，而使黑客能够进一步改变数据的访问权限。否则，软件执行过程中"被污染"的数据范围不会因为误差的引入而发生变化。例如，当浮点数值被转换成某一整型的指针时，由于误差而使得指针位置发生变化，从而使软件的数据依赖发生变化。因此，软件中浮点数值向整数进行传播是本文分析数据依赖变化的关键点。而当该传播可能引起传播后整数值发生变化时，我们即可构建相应的攻击输入，以便在后续步骤中进一步验证是否会真实存在安全漏洞隐患。

我们重点分析软件中浮点数值向整数值的传播。本文分析方法首先标定软件中各变量的数据依赖，并以此来搜索依赖于浮点数值的整数变量 *i*，与此同时，我们通过与算法 1 类似的方法获得变量 *i* 的数值变量符号表达式 g，以及 *round(g)*的符号式。最终，我们通过构建约束来获得攻击输入。所构建的攻击约束 *attcond* 为：

$$|round(g)| > 0.5 \tag{5}$$

同理，当攻击约束 *attcond* 在约束求解器中可解时，我们直接解得攻击的输入值，而当攻击约束 *attcond* 不可直接求解时，我们通过随机搜索方式搜索获得满足条件的攻击输入。

3.3 高精度动态攻击验证

由于静态分析本身的技术限制，其输出可能会存在误报(false positive)问题。我们可以通过高精度动态验证的方式来消除误报，获得最终的安全漏洞隐患。高精度动态验证首先将原始待测软件中的每一个浮点数值转换成运行时分配内存的高精度版本。通过大内存计算这些数值程序会大大减少每一步的舍入误差，且由于可以在运行时动态增加每一个浮点数值的表示内存，我们可以通过以不同的内存大小多次运算来动态判断当前各变量的执行精度。当执行精度不够时，我们可以通过增加内存来动态提升精度，获得需要的结果。因此，我们可以保证高精度数值计算在一定有效数字的范围内就是实数计算的结果。

本文通过动态比对待测试软件与对应的高精度软件版本来验证第二阶段分析获得的攻击输入是否有效。当我们以第二阶段的静态分析方法获得了软件的攻击输入之后，以该输入同时驱动待测试软件与对应的高精度软件版本执行，并在执行过程中不断比对对应的软件行为是否存在不同。如果对应的软件行为不同，则意味着误差引起了软件行为的改变，检测到的安全漏洞隐患即为真实的漏洞隐患。反之，如果因为静态分析的精度不够，对应的软件行为仍完全一致，则意味着实际运行时误差积累并未使软件行为发生变化，对应的安全漏洞隐患为误报，会在输出前被过滤。

本文基于软件剖析技术(Software Profiling)来比对待测软件与对应高精度软件在软件行为上的不同点。软件剖析会在当前软件源代码中的每一个条件约束与数值依赖的整数变量位置插装一条记录语句，记录当前的位置标号，条件约束的取值，整数变量的取值，以及内存中的关键数据对象。并在高精度软件的相同位置插装记录对应信息的记录语句。这里的位置标号用于追踪软件在当前输入下的执行路径，当待测软件与高精度软件在执行路径上完全一致时，我们认为软件的误差积累没有引起控制流发生变化。而软件的数据依赖信息则通过在软件剖析中记录的其它信息来分析获取。

这里我们引入了污点分析(Taint Analysis)来获得软件剖析所需记录的关键内存数据对象，并以这些数据对象作为误差积累是否引起数据依赖变化的标准。通过污点推进的方式，污点分析静态定位了输入所能影响到的内存区域，对于不受输入影响的变量，以及指针所指向的内存对象，在我们的检测方法中被称为保护区。本文的动态验证方法直接将和保护区相关的变量作为关键内存对象的搜索来源。对于类型为指针的相关变量，本文的软件剖析会进一步序列化指针所指向的内存对象，并将整个对象值作为数据依赖比对的依据。当待测软件与对应高精度软件的关键内存对象在相同的测试输入下存在结构上的不同 (例如：数组大小不同，单链表长度不同等等)或者非数值类型的变量值差异(例如：int,char,bool 等类型的数值差异)时，我们判定软件的数据依赖因为数值误差的引入而发生的变化。所检测到的安全漏洞隐患也即得到确认。

4 实例研究

本文在 9 个著名的开源项目上进行了实验，分析代码中是否存在安全漏洞隐患，通过这些实例研究，我们希望能够回答以下三个重要的研究问题：

研究问题 1: 数值稳定性相关的安全漏洞隐患在软件中是否普遍存在?

研究问题 2: 本文提出的动静态相结合的检测方法是否适用于大型软件的漏洞隐患检测?

研究问题 3: 数值误差要积累到怎样的程度才会造成软件中存在稳定性相关的安全漏洞隐患?

4.1 实验设置

我们实现了数值稳定性相关安全漏洞隐患检测方法的工具原型，其中数值变量符号式提取模块是基于 KLEE 1.3.0 开发的，静态攻击流程分析基于 ROSE compiler 0.9.5a 开发，动态验证模块基于 iRRAM 2013.01 开发[17]。实验对象以 gcc 5.4.0 编译，并通过 Python2.7.10 来驱动测试。整体的测试环境运行在一台 CPU 为 Intel i5-5257U 2.7GHz*2 的双核处理器，内存 8GB 的笔记本计算机上，操作系统为 Ubuntu14.0.1。

我们以 9 个业界广泛使用的开源项目作为实验对象，其具体情况如表 1 所示。为了检验本文检测方法的

有效性，降低因实验对象的偶然因素带来的偏差，我们尽量选择了不同类别的大规模开源项目的最新版本。其中最大实验对象的代码行数超过了 700 万行，软件包大小为 1.42GB，最小的实验对象也达到了 50 万行的代码规模，软件包大小为 36.7MB。

Table 1 Details of our Experimental Objects

表 1 实验对象详细信息

项目	大小	源代码文件数	源代码总行数	版本号	项目类别
Blender	189.9MB	2,752	1,745,612	2.78c	游戏引擎
Clang	121.5MB	6,821	1,396,611	3.9.1	编译器
Crystal Space	453.2MB	1,149	844,094	2.0	游戏引擎
Firefox	1.42GB	12961	7,187,165	50.0.1	浏览器
MPlayer	109.1MB	2,746	1,275,858	1.3.0	视频播放器
Mysql	429MB	2,564	2,425,082	5.7.17	数据库
Nebula Device 2	48.6MB	1,246	576,081	vc8	游戏引擎
OpenSceneGraph	36.7MB	1,851	512,536	3.4	游戏引擎
PHP	167.1MB	1,027	1,168,792	7.1.0	网页开发语言

为了对各项目中的数值相关代码进行快速检索，提高本文方法的检测效率。在实施本文的安全漏洞检测之前，我们通过静态预处理标定了各项目中的数值代码片段。当项目源代码中存在由浮点数计算的条件判断或由浮点数参与的强制类型转换时，对应的浮点数上下文操作即被标定为一个数值代码片段。经过标定的数值代码片段集即为本文软件行为变化的候选分析区来实施本文三阶段的安全漏洞隐患检测。

当我们检测到实验对象中存在由数值稳定性引起的控制流路径变化时，如果对应路径条件在循环中，我们会进一步通过后处理来检测循环相关的拒绝服务(Denial-of-Service, DoS)安全漏洞隐患。即检测是否存在攻击输入使该循环卡死。当存在这样的攻击输入时，黑客可以反复尝试这样的攻击输入，使当前项目中各个线程均陷入这样的死循环中，直至系统资源耗尽而拒绝再为客户端提供服务为止。

当循环中存在因误差积累而引起的数值稳定性相关拒绝服务漏洞隐患时，误差积累会造成循环条件在当前输入下始终为真(或始终为假)。此时我们进一步分析攻击输入在循环迭代后是否会因为误差，而在循环条件的关系表达式中形成不动点，或者发生反向变化，即可判断是否存在循环相关的拒绝服务(Denial-of-Service, DoS)安全漏洞隐患。这里的反向变化是指在不考虑误差时，规范化关系表达式的左部随着循环迭代的次数的增加而变小(或变大)，而考虑误差时对应的关系表达式左部随着循环迭代的次数的增加而变大(或变小)。

4.2 实验结果

表 2 列出了本文检测方法在 9 个开源项目上的实验检测结果，我们在 9 个开源项目共 4848 个数值代码片段中共发现 77 个数值稳定性相关的安全漏洞隐患。从表 2 的数值代码片段标定数量可以看出，Blender 中含有的数值代码最多，共有 2056 个数值代码片段，而 Crystal Space 中含有的数值代码量最少，仅检测到 82 个数值代码片段。无论代码的规模大小，各实验对象或多或少地都存在一些数值稳定性相关的软件安全漏洞隐患。由此可知，数值稳定性相关的安全漏洞隐患目前在软件中是普遍存在的，且安全漏洞隐患的数量基本和数值代码片段的数量正相关，其线性相关系数为 0.903。另外，对于每一个实验对象，我们检测到因数值稳定性引起的控制流路径变化数量均高于因数值稳定性引起的数据依赖变化数量，我们检测到 77 个数值稳定性相关的安全漏洞隐患中有 62 个是控制流路径的变化，而仅有 15 个是数据依赖的变化。因此，数值稳定性

引起的控制流变化相对于数据依赖变化来说更为普遍。

Table 2 Numbers of Detected Vulnerabilities related to Numerical Stability

表 2 数值稳定性相关安全漏洞隐患的检测结果数量

项目	数值代码片段的位置标定数量	数值稳定性相关的漏洞隐患总数	数值稳定性引起数据依赖的变化数量	数值稳定性引起控制流路径的变化数量	数值稳定性引起循环相关的 DoS 漏洞隐患数量
Blender	2056	24	5	19	7
Clang	486	1	0	1	1
Crystal Space	82	4	1	3	0
Firefox	1070	17	4	13	4
MPlayer	471	10	2	8	0
Mysql	141	2	0	2	0
Nebula Device 2	166	8	1	7	0
OpenSceneGraph	267	8	2	6	0
PHP	109	3	0	3	2
总和	4848	77	15	62	14

我们进一步分析了实验对象中由数值稳定性引起的循环相关 DoS 漏洞隐患。作为由数值稳定性引起控制流变化的特例,我们共在 4848 个数值代码片段中发现 14 个由数值稳定性引起的循环相关 DoS 漏洞隐患(如表 2 所示)。表 3 给出了这 14 个数值稳定性引起的循环相关 DoS 漏洞隐患的具体情况。我们手工确认了每一个漏洞隐患，在表 3 中列出了利用对应漏洞的攻击输入，以及攻击发生时的相对误差。另外，我们采用了文献[25]中使用的启发式数值程序测试用例生成方法，为每一个数值输入生成了 460 个测试用例来进行测试，并在表 3 中列出了观测到的最大相对误差及对应的测试输入。

由表 3 列举的数据可知，产生 DoS 漏洞隐患的数值误差往往很小，在我们的实验中其相对误差均小于 3.00E-9，最小的情况仅需要 7.39E-42 的相对误差就可以造成数值程序存在安全漏洞。不仅如此，对应的攻击输入也往往并不在使数值误差累积到最大值的时候出现。从表 3 中我们可以看到，用例中观测到的最大相对误差往往大于漏洞被利用时的相对误差。由此可见，很小的数值误差累积就可能会造成软件中存在相关的安全漏洞隐患，传统的数值稳定性分析方法在检测相关的安全漏洞隐患时会因为仅关注较大的数值误差累积而失效。

值得注意的是，本文方法所检测到的漏洞隐患可能并不一定会被确认为当前版本的软件漏洞，但一般来说都值得软件开发人员进一步关注。例如：某软件的后台业务模块中存在因数值稳定性问题而引起的控制流变化或者数据依赖变化，但由于前台模块的功能尚未完善，黑客在当前版本软件中并不能在已有功能点中找到攻击方案，因此并不能成为当前版本的软件漏洞。随着软件版本升级和前台模块功能的加强，在下一版本中这一隐患就可能会演变成实际的漏洞。对于本文实验中检测到的漏洞隐患，我们正在同相关人员合作进行进一步的确认追踪。

Table 3 Loop related DoS Vulnerability Hazards

表 3 循环相关的 DoS 漏洞隐患

漏洞隐患所在项目	源代码文件名	漏洞代码段起始行号	漏洞被利用时的相对误差(攻击输入)	观测到的最大相对误差(对应输入)
Blender	key.c	662	8.13E-10(1.23E09)	1.46E-08(1.67E03)
Blender	key.c	995	1.61E-12(6.23E11)	3.82E-08(2.43E01)
Blender	shadeoutput.c	104	1.72E-14(5.82E13)	1.27E-08(2.47E05)
Blender	scaling.c	946	1.19E-13(8.41E12)	3.05E-08(7.83E01)
Blender	scaling.c	1077	7.09E-13(1.41E12)	3.26E-08(1.35E00)
Blender	interface_draw.c	1275	6.76E-11(1.48E08)	1.06E-08(3.47E02)
Blender	node_draw.c	687	4.05E-10(2.47E07)	1.12E-09(6.79E00)
Clang	shortest-path-suppression.c	12	2.14E-09(4.67E08)	4.27E-09(2.34E08)
Firefox	cairo-stroke-style.c	245	7.09E-13(1.41E12)	1.86E-08(5.39E07)
Firefox	audio_encoder_opus_unittest.cc	28	6.76E-21(1.48E18)	4.26E-09(2.35E06)
Firefox	pixman-conical-gradient.c	10	1.17E-19(5.39E19)	3.13E-17(8.58E01)
Firefox	pixman-conical-gradient.c	13	8.50E-22(7.39E21)	1.79E-18(1.00E13)
PHP	gd_rotate.c	518	6.36E-19(5.66E20)	4.49E-17(6.48E02)
PHP	gd_rotate.c	522	7.39E-42(4.87E43)	3.15E-17(8.87E05)

5 漏洞修复途径

本节主要讨论对于数值稳定性引发安全漏洞的修复问题。一般来说，当采用本文方法找到相关安全性漏洞后，可以考虑以下三种修复途径：

一.通过采用更加稳定的数值算法来修复该类漏洞。很多情况下，开发人员在软件中由于采用了不稳定的数值算法，从而使软件存在安全漏洞，这种情况下采用更加稳定是数值算法显然是修复该类漏洞的最佳手段。例如,对于线性方程组的求解问题采用列主元素消去法显然会比原始的高斯消去算法在稳定性上更具有优势。该修复手段通过解决数值稳定性问题，而从根源上修复该类漏洞。

二.通过解耦黑客的攻击路径，使其与相应的数值计算相互独立来修复该类漏洞。有时候软件中的安全漏洞并不是由于开发人员采用了不稳定的数值算法而引起的，或者由于某些原因不能修改软件中的数值计算过程。此时可以考虑将黑客可能的攻击路径与数值计算逻辑解耦，使其与软件中的数值计算过程相互独立，这样即使软件的数值计算部分仍然存在误差，也不再会使软件存在安全性漏洞。

三.通过提高数值计算的精度来修复该类漏洞。直接提高数值计算的精度是减少数值误差的有效手段，也会成为修复数值稳定性引发安全漏洞的有效手段。然而这一方法常常以增加计算负载，消耗更多的计算资源为代价。因此我们建议将该方法作为修复软件安全性漏洞的备用途径，在可以通过其它方法修复漏洞时，尽量采用其它的修复途径。

6 相关工作

本文给出了一种数值稳定性相关安全漏洞隐患的自动化检测方法，它能够帮助设计开发人员检测软件中由于数值稳定性而引入的安全漏洞隐患。由于目前并没有针对数值稳定性安全漏洞检测方面的研究，这里我们主要调研了数值稳定性自动化分析的相关研究。

[32][29][33]长期以来，数值计算作为一个专门的研究领域，在探讨误差积累与结果偏差的关系，寻求尽可能减少误差积累的数值算法等方面取得了许多重要的研究成果[6-8,26,27]。近年来，软件工程领域出现了一系列数值稳定性的自动化分析技术，以解决软件规模与复杂性增加，难以人工进行数值稳定性分析等问题。例如：Putot 和 Goubault 等学者基于仿射算术法则(affine arithmetic)来分析软件可能达到的最大误差积累[22]，并提出了基于这项技术的静态程序分析[28]以及抽象解释方法[29]，使分析精度大大提升。Benz 等学者给出了一套浮点数值分析与调试工具：FPDebug，该工具通过基于二进制操作的轻量级程序切片(Program Slicing)技术来发现软件中的数值错误[9]。Bao 和 Zhang 等学者进一步通过追踪容易出错的浮点数值来提高数值错误的发现概率[30]。Boldo 等学者从形式化验证的角度来自动分析 C 程序的浮点操作性质[19,20]。 Darulova 和 Kuncak 等学者针对数值稳定性问题给出了一套类型系统[31]，以便通过类型系统的推理来推断实际的误差积累。Martel 提出了数值误差的前向推导语义[32]，并基于这一语义开发了一个优化器[33]，以便将程序优化成精度更高的算法[18]。Monniaux 针对不同硬件平台与编译器条件下浮点操作在语义上的弱点给出了验证方法[34]。在测试用例生成方面，Miller 和 Spooner 等学者提出了一种称为数值最大化法则的搜索技术[35]，并以此来制导测试用例的生成。Bagnara 等学者基于启发式搜索来求解混合执行(concolic testing)中带有浮点数值的符号约束[36]。Barr 等学者针对类似于上溢出与下溢出等浮点数异常来检测符号执行中的浮点错误[10]。更进一步，Zou 等学者通过遗传算法来改进 FPDebug 的能力，并能自动找到使结果发生大量偏差的程序输入[37]。许多近年来的研究工作采用随机采样方法来分析数值程序的稳定性，其中有代表性的方法称为 CESTAC[11]，它通过在数值计算中引入随机进位模式来估测其误差积累。Vignes 等学者基于 CESTAC 算法提出了离散随机算术，并构建了可用的数值计算分析库 CADNA[21]。Parker 等学者对蒙特卡洛算术(Monte Carlo Arithmetic, MCA)进行了形式化[38]，并以此作为随机数值稳定性分析的理论基础。稍后，Eggert 和 Parker 等学者基于这一理论构建了工具 wonglediff[39]，在硬件 FPU 层面对误差进行估计。本文作者也参与了数值稳定性自动分析技术的相关研究，在已有工作的基础上引入了表达式的随机变换，并以此来分析与诊断软件的数值稳定性[25,40]。以上这些工作主要关注于数值计算本身的误差积累程度，以及运算结果的正确性。但它们仍然未能解决数值稳定性引起的软件安全性问题，本文针对这一新问题给出了有效的解决方案。

7 结束语

安全漏洞检测是保障软件安全性的重要手段之一，在软件安全领域受到广泛关注。随着互联网的发展，黑客的攻击手段日趋多样化，且攻击技术不断翻新，使软件安全受到了新的威胁。本文描述了当前软件中实际存在的一种新类型的安全漏洞隐患，我们称之为数值稳定性相关的安全漏洞隐患。利用这种新类型的漏洞，黑客可以绕过现有的软件安全防护措施而对系统实施攻击，且已有的数值稳定性分析方法主要关注数值误差的极大值点，很难检测到误差累积较少就会引入的安全漏洞，因而这一新类型的安全漏洞隐患十分危险。

面对这一挑战，本文首先从数值稳定性引起软件行为改变的角度对数值稳定性相关的安全漏洞隐患给出了明确定义，并提出了相应的自动化检测方法。该方法基于动静态相结合的程序分析与符号执行技术，通过数值变量符号式提取、静态攻击流程分析、以及高精度动态攻击验证三个步骤，来检测和分析软件中可能存在的数值稳定性相关安全漏洞。我们在业界多个著名开源软件上进行了实例研究，实验结果表明， 数值稳定性相关的安全漏洞隐患与实际软件中数值误差的积累程度无关，很多误差积累很小的模块也同样会存在安全漏洞隐患。正因为如此，数值稳定性相关的安全漏洞隐患在许多含有数值计算代码的实际软件中普遍存在，且安全漏洞隐患的数量同软件中数值计算的代码量正相关。另外，实验结果也说明了本文方法能够有效检测

到实际大型软件中真实存在的数值稳定性相关安全漏洞隐患。

References:

[1] Symantec. Internet Security Threat Report, 2012, p. 52. https://www.symantec.com/content/dam/symantec/docs/reports/istr-17-2012-en.pdf

[2] Symantec. Internet Security Threat Report, 2017, p. 77. https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf

[3] Wassermann G, Su Z. Static Detection of Cross-site Scripting Vulnerabilities. In: Proceedings of the 30th International Conference on Software Engineering. New York, NY, USA: ACM, 2008. 171–180.

[4] Møller A, Schwarz M. Automated Detection of Client-state Manipulation Vulnerabilities. In: Proceedings of the 34th International Conference on Software Engineering. Piscataway, NJ, USA: IEEE Press, 2012. 749–759.

[5] Zheng Y, Zhang X. Path Sensitive Static Analysis of Web Applications for Remote Code Execution Vulnerability Detection. In: Proceedings of the 2013 International Conference on Software Engineering. Piscataway, NJ, USA: IEEE Press, 2013. 652–661.

[6] Higham NJ. Accuracy and stability of numerical algorithms[B]. 2nd edition ednSociety for Industrial and Applied Mathematics, 2002, p. 680.

[7] Miller W. Software for Roundoff Analysis. ACM Trans. Math. Software, 1975,1:108–128.

[8] Wilkinson JH. Rounding Errors in Algebraic Processes[B].Dover Publications, Incorporated, 1994.

[9] Benz F, Hildebrandt A, Hack S. A dynamic program analysis to find floating-point accuracy problems. SIGPLAN Not., 2012,47:453-462.

[10] Barr ET, Vo T, Le V, Su Z. Automatic detection of floating-point exceptions. In: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. New York, NY, USA: ACM, 2013. 549–560.

[11] Vignes J. A stochastic arithmetic for reliable scientific computation. Mathematics and Computers in Simulation, 1993,35:233-261.

[12] Zhao S. A reliable computing algorithm and its software application ISReal for arithmetic expressions. Science China Information Sciences, 2016,46:698-713 (in Chinese with English abstract). doi:10.1360/N112015- 00061

[13] NVD. CVE-2010-4467, 2010. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4467

[14] NVD. CVE-2010-4645, 2011. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4645

[15] Chen G, Jin H, Zou D, Dai W. On-Demand Proactive Defense against Memory Vulnerabilities. In: Network and Parallel Computing. Springer, Berlin, Heidelberg, 2013. 368-379.

[16] Caballero J, Grieco G, Marron M, Nappa A. Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. New York, NY, USA: ACM, 2012. 133–143.

[17] Müller NT. The iRRAM: Exact Arithmetic in C++. Lecture Notes in Computer Science, 2001,2064(1):222-252.

[18] Martel M. Program transformation for numerical precision. In: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation. Savannah, GA, USA: ACM, 2009. 101-110.

[19] Boldo S, Filliatre JC. Formal Verification of Floating-Point Programs. In: 18th IEEE Symposium on Computer Arithmetic, 2007. ARITH '07. 2007. 187-194.

[20] Boldo S, Melquiond G. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In: 2011 20th IEEE Symposium on Computer Arithmetic (ARITH). 2011. 243-252.

[21] Jézéquel F, Chesneaux JM. CADNA: a library for estimating round-off error propagation. Computer Physics Communications, 2008,178:933-955.

[22] Putot S, Goubault E, Martel M. Static Analysis-Based Validation of Floating-Point Computations. In: Alt R, Frommer A, Kearfott RB, Luther W eds. Numerical Software with Result Verification. Springer Berlin Heidelberg, 2004. 306-313.

[23] Wang K, Zhang Y, Liu P. Call Me Back!: Attacks on System Server and System Apps in Android Through Synchronous Callback. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: ACM, 2016. 92–103.

[24] Wang L, Li F, Li L, Feng XB. Principle and practice of taint analysis. Ruan Jian Xue Bao/Journal of Software, 2017,28(4):860−882 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5190.htm

[25] Tang E, Barr E, Li X, Su Z. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In: Proceedings of the 19th international symposium on Software testing and analysis. New York, NY, USA: ACM, 2010. 131-142.

[26] Miller W. Toward Mechanical Verification of Properties of Roundoff Error Propagation. In: Proceedings of the Fifth Annual ACM Symposium on Theory of Computing. New York, NY, USA: ACM, 1973. 50–58.

[27] Miller W, Spooner D. Software for Roundoff Analysis, II. ACM Trans. Math. Software, 1978,4:369–387.

[28] Goubault E. Static Analyses of the Precision of Floating-Point Operations. In: Proceedings of the 8th International Static Analysis Symposium. 2001. 234–259.

[29] Goubault E, Putot S. Static Analysis of Numerical Algorithms. In: Proceedings of the 13th International Static Analysis Symposium. 2006. 18–34.

[30] Bao T, Zhang X. On-the-fly Detection of Instability Problems in Floating-point Program Execution. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. New York, NY, USA: ACM, 2013. 817–832.

[31] Darulova E, Kuncak V. Trustworthy Numerical Computation in Scala. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. New York, NY, USA: ACM, 2011. 325–344.

[32] Martel M. Propagation of Roundoff Errors in Finite Precision Computations: A Semantics Approach. In: Programming Languages and Systems. 2002. 159-186.

[33] Martel M. Semantics-Based Transformation of Arithmetic Expressions. In: Static Analysis. 2007. 298-314.

[34] Monniaux D. The Pitfalls of Verifying Floating-point Computations. ACM Trans. Program. Lang. Syst., 2008,30:12:1–12:41.

[35] Miller W, Spooner DL. Automatic Generation of Floating-Point Test Data. IEEE Transactions on Software Engineering, 1976,SE-2:223-226.

[36] Bagnara R, Carlier M, Gori R, Gotlieb A. Symbolic Path-Oriented Test Data Generation for Floating-Point Programs. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST). 2013. 1-10.

[37] Zou D, Wang R, Xiong Y, Zhang L, Su Z, Mei H. A Genetic Algorithm for Detecting Significant Floating-Point Inaccuracies. In: Proceedings of 37th International Conference on Software Engineering. 2015. 20-22.

[38] Parker DS, Pierce B, Eggert PR. Monte Carlo arithmetic: how to gamble with floating point and win. Computing in Science & Engineering, 2000,2:58-68.

[39] Eggert PR, Parker DS. Perturbing and evaluating numerical programs without recompilation: the wonglediff way. Software Practice and Experience, 2005,35:313−322.

[40] Tang E, Zhang X, Muller N, Chen Z, Li X. Software Numerical Instability Detection and Diagnosis by Combining Stochastic and Infinite-precision Testing. IEEE Transactions on Software Engineering, 2017

附中文参考文献:

[12] 赵世忠.算术表达式的一种可信计算算法及其软件 ISReal.中国科学:信息科学,2016,46:698-713. doi:10.1360/N112015-00061

[24] 王蕾,李丰,李炼,冯晓兵.污点分析技术的原理和实践应用.软件学报,2017,28(4):860−882. http://www.jos.org.cn/1000-9825/5190.htm