多线程 Java 程序安全行为模型的静态检查方法

金 英 李泽鹏 张 晶 刘 磊
(吉林大学计算机科学与技术学院 长春 130012)

摘 要 多线程作为支持程序结构化和并行化的重要机制,其应用越来越广泛,多线程应用程序的安全性也成为新的研究热点之一.针对 Java 多线程程序,文中采用参数化扩展上下文无关文法作为其安全相关行为模型的抽象表示,给出了从多线程 Java 程序自动生成安全相关行为模型的方法,形式地描述了静态检查该模型是否满足安全策略的实现,并应用到携带模型代码方法的实现框架中.该方法为安全执行非信任多线程 Java 移动代码提供了有效支持.

关键词 多线程 Java 程序;安全相关行为模型;静态检查;参数化扩展上下文无关文法
中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2009.01856

Static Checking of Security Related Behavior Model for Multithreaded Java Programs

JIN Ying LI Ze-Peng ZHANG Jing LIU Lei
(*College of Computer Science and Technology, Jilin University, Changchun* 130012)

Abstract Multithreading is an important mechanism for supporting program structuring and parallel computation. With the wide usage of multithreading, security for multithreaded application has become one of new hot research topics. This paper focuses on the security of Java multithreaded programs. At first, parameterized extended context free grammar has been used to formally represent security related behavior model for multithreaded Java program; then the way of automatic generation of such model is introduced, and an approach to statically check security related behavior model is formalized. The method has been applied in the framework of model carrying code. It has been indicated that the method provides effective support for safe execution of untrusted multithreaded Java mobile code.

Keywords multithreaded Java program; security related behavior model; static checking; parameterized extended context free grammar

1 引 言

当今社会很多领域越来越多地依赖于软件系统来完成敏感的、重要的、甚至是关键性的功能,伴随着软件系统复杂度的不断提高,其中的缺陷和漏洞也难以避免.特别是随着互联网的引入,使得系统入侵、网络攻击、信息泄露和数据丢失等安全问题日益突出,软件安全性成为一个不容忽视的问题.软件安全性研究的重点是建立可以降低或者避免软件安全性错误的过程、方法和工具.所谓的软件安全性错误指的是在软件开发过程中产生的、导致软件执行违

收稿日期:2009-04-19;最终修改稿收到日期:2009-08-05.本课题得到国家自然科学基金青年基金(60603031)资助.**金 英**,女,1971 年生,博士,副教授,主要研究方向为软件工程、移动代码安全、软件形式化. E-mail: jinying@jlu.edu.cn. **李泽鹏**,男,1984 年生,博士研究生,主要研究方向为移动代码安全、软件工程. **张 晶**,女,1975 年生,博士,讲师,主要研究方向为软件形式化、程序分析. **刘 磊**,男,1960 年生,教授,博士生导师,主要研究领域为程序分析、语义网和软件形式化.

反安全策略的所有错误.其中对于软件源代码的安全性分析和检查是检测安全性错误最直接和有效的途径之一[1].

关于顺序程序的安全性分析、检查和形式验证方法已有很多研究,产生了一批研究成果和支持工具,而关于并发程序的安全性分析研究则相对较少[2]. 多线程作为支持程序结构化和并行化的重要机制,其应用越来越广泛,而针对多线程应用程序的安全性分析和检查则成为新的研究热点之一.近几年来,Java 语言以其简单性、面向对象、分布式、平台无关、可移植、多线程、动态性等特点成为新的、流行的、跨平台、适合于分布式计算环境的面向对象编程语言,研究多线程 Java 程序的安全性分析和检查方法十分必要.

面向源程序的安全性分析和检查方法主要分为静态方法和动态方法两大类,以模型检查、静态分析、符号执行和动态检测等为代表[3].动态方法主要是基于执行程序获得的相关信息来检查是否满足安全特性要求,而静态方法则主要是通过对源代码的分析,建立属性相关的抽象模型,进而检查或者验证是否满足安全策略.动静态方法各有利弊,而当前一种趋势是把二者结合起来[3-4].

本文给出了通过静态分析多线程 Java 程序来进行安全性检查的方法,主要考察的是多线程 Java 程序的安全相关行为.本文中安全性检查的含义是:针对用户给出的、定义在安全相关操作集合上的安全策略,检查程序的所有可能执行是否会违反对应的安全策略.为了能够通过静态分析自动检查多线程 Java 程序的安全性,需要建立该程序的安全相关行为模型(即能够代表该程序所有可能的执行产生的所有安全相关操作序列),并且与安全策略的形式定义相比较,从而确定该程序是否满足安全策略.首先,我们提出一种基于扩展上下文无关文法的安全相关行为模型的形式定义,并给出由 Java 源程序自动生成其安全相关行为模型的方法;接下来,引入安全相关行为模型静态检查方法,采用抽象描述给出了该方法的实现,其主要思想是基于安全相关行为模型生成所有可能的安全相关操作序列,并与基于扩展有限状态机(EFSA)的安全策略形式定义相比较,进而检查该程序的所有执行路径是否满足安全策略.特别地,针对静态分析无法确定程序中循环次数的缺点,给出了简单的解决方法.最后,将本文方法应用于一种安全执行非信任移动代码框架——携带模型代码 MCC 方法中[5],为在 Java 平台上实现 MCC 方法提供了有效的支持.

2　安全策略定义

目前有各种不同的安全策略形式定义方法,较常用的有:(1)基于各种状态机的形式定义方法,包括有限状态机、正则表达式、下推自动机和扩展有限自动机等;(2)基于逻辑的定义方式,如采用谓词表达式等.

本文将使用文献[6]中的 SPDL 形式规范语言.该语言是在基于事件的正则表达式(REE)和行为监控规格语言(BMSL)基础上扩展得到的,通过对其分析可以转换成相应的扩展有限状态自动机.

定义 1.　扩展有限状态自动机(EFSA)是一个七元组$(Q,q_0,F,V,Env,\Sigma,\delta)$,其中

Q 表示有限状态集合;

q_0 是初始状态;

F 是终止状态集合;

V 是状态变量的有限集合;

Env 是所有可能环境的集合,每个环境为状态变量可能的取值情况,变量值的改变意味着环境的更新;

Σ 是输入符集合,这里代表安全相关操作集合;

δ 是转换函数:$Q\times\Sigma\times Env \Rightarrow 2^{(Q\times Env)}$.

可以看出,EFSA 是一个不确定的状态机.

图 1 给出的就是分别用 BMSL 和 EFSA 定义的一个简单安全策略例子(类似于文献[5]中的 Figure 7),表示如果一个应用程序在读了关键目录下的文件后再进行网络访问则违反了该安全策略.其中 FileRead 表示所有文件读相关的安全操作集合(如 BufferedReader.readline 等),sensitive(f)表示该文件是关键性的,这里是指在目录"/home/sensitivefiles/"下的所有文件,而 socket 指的是所有导致 socket 调用的操作,any 指的是除了 FileRead 和 socket 相关操作以外的任意操作.

定义 2.　设 $E=(Q,q_0,F,V,Env,\Sigma,\delta)$ 是一个

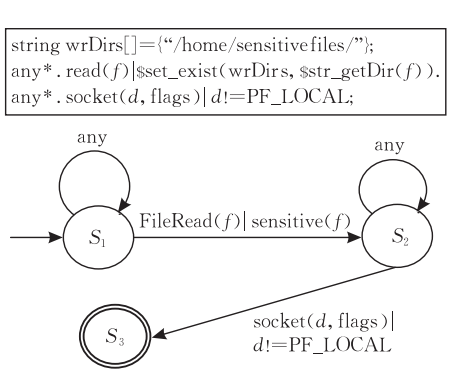

图 1　一个简单安全策略例子

扩展的有限自动机，对于给定的一个符号串 $\alpha=a_1,\cdots,a_n\in\Sigma^*$ 以及一个初始环境 σ_0，称该符号串 α 是 E 所接受的，如果存在由状态-环境对构成的序列 $[(q_1,\sigma_1),\cdots,(q_n,\sigma_n)]$，使得

$(q_1,\sigma_1)\in\delta(q_0,\sigma_0,a_1)\cdots(q_n,\sigma_n)\in\delta(q_{n-1},\sigma_{n-1},a_n)$.

通常一个安全策略声明的是所有违反安全策略的操作序列情形，因此，定义一个安全策略的扩展有限自动机给出的是所有该安全策略不允许的操作序列集合.

3 多线程 Java 程序安全相关行为模型

本节首先给出参数化扩展上下文无关文法及其所接受语言的形式定义，然后介绍如何基于参数化扩展上下文无关文法来定义一个多线程 Java 程序的安全相关行为模型，以及从多线程 Java 源代码如何自动生成相应的安全相关行为模型，并且针对循环和递归问题给出简单的处理方法.

3.1 参数化扩展上下文无关文法

定义 3. 一个参数化扩展上下文无关文法 G 是一个五元组 (V_N,V_T,S_0,α,PS)，其中

V_N 是一个有限的非终极符集合；

V_T 是一个有限的终极符集合；

S_0 是开始符，$S_0\in V_N$；

α 是一个元数函数，它的定义域是 $V_N\cup V_T$，值域是整数，对于一个符号 $a\in V_N\cup V_T$，$\alpha(a)$ 称为符号 a 的元数(arity)，即代表每个符号的参数个数；

PS 是一个有限的产生式集合，每个产生式的形式是 $a_0(X_0)\rightarrow\boldsymbol{RE}(a_1(X_1),\cdots,a_n(X_n))$，其中

$a_0\in V_N$，$a_i\in V_N\cup V_T(1\leq i\leq n)$，$X_0$ 是 $\alpha(a_0)$ 元组的形参；

X_i 是 $\alpha(a_i)$ 元组的实参，$1\leq i\leq n$；

$\boldsymbol{RE}(a_1(X_1),\cdots,a_n(X_n))$ 是定义在 $\{a_1(X_1),\cdots,a_n(X_n)\}$ 上的正则表达式.

参数化扩展上下文无关文法允许符号带参数，产生式左边的符号对应的是形参，产生式右边的符号对应的实参表达式；而且产生式的右部可以是正则表达式(即允许出现星闭包)，这些特点为表示 Java 程序的安全相关行为模型提供了很好的支持.

一个简单的参数化扩展上下文无关文法 $G_e=(V_N,V_T,S_0,\alpha,PS)$，定义如下：

$V_N=\{S,A\}$；

$V_T=\{a,b\}$；

$S_0=S$；

$\alpha=\{S\rightarrow 2,A\rightarrow 1,a\rightarrow 2,b\rightarrow 1\}$；

$PS=\{S(x,y)\rightarrow b(x)*A(y)$；

$A(x)\rightarrow a(x,2)$

$\}$.

定义 4. 设有一个参数化的扩展上下文无关文法 $G=(V_N,V_T,S_0,\alpha,PS)$ 以及一个符号串 $u=a_1(Y_1)\cdots a_n(Y_n)$，其中 $\{a_1,\cdots,a_n\}\subseteq(V_N\cup V_T)$，我们称 u 推导出 v，记作 $u\Rightarrow_p v$，如果存在一个产生式 $a_i(X)\rightarrow w$，满足

$v=a_1(Y_1)\cdots a_{i-1}(Y_{i-1})w'a_{i+1}(Y_{i+1})\cdots a_n(Y_n)$，

其中 w' 是将 w 中的参数 X 替换成 Y_i 而得到的.

$\Rightarrow_p^*$ 被用来表示零次或者多次推导.

针对参数化扩展上下文无关文法 G_e 的定义，存在如下推导：

$S(x,y)\Rightarrow_p b(x)*A(y)\Rightarrow_p b(x)*a(y,2)$.

定义 5. 给定一个参数化扩展上下文无关文法 $G=(V_N,V_T,S_0,\alpha,PS)$ 和一个 $\alpha(S_0)$ 元组参数 Y，我们用 $L(G,Y)$ 来表示 G 在参数 Y 下的符号串集合(语言)，则

$L(G,Y)=\{L(a_1(X_1)\cdots a_n(X_n))\mid$

$S_0(Y)\Rightarrow_p^* a_1(X_1)\cdots a_n(X_n)\wedge\{a_1,\cdots,a_n\}\subseteq V_T\}$，

其中 $L(a_1(X_1)\cdots a_n(X_n))$ 表示正则表达式 $a_1(X_1)\cdots a_n(X_n)$ 所接受的语言.

可以看出，一个参数化扩展上下文无关文法所接受的语言是由其定义以及初始符号的实参序列决定的，其组成是所有由该参数化扩展上下文无关文法开始符推导出的、只包含终极符的正则表达式所接受的语言的并集.

针对参数化扩展上下文无关文法 G_e 的定义，有

$L(G_e,(m,1))=L(b(m)*a(1,2))=$

$\{a(1,2),b(m)a(1,2),b(m)\ b(m)\ a(1,2),\cdots\}$.

3.2 安全相关行为模型定义方法

对于 Java 语言来讲，安全相关行为(操作)可以建立在不同层次：用户级的方法调用、API 方法调用或者系统调用. 如果建立在用户应用程序的方法调用级别上，则安全相关行为模型只需要从用户定义的类和方法中抽取，每个安全相关操作的形式为 $\langle C,o,m\rangle(X)$，表示应用程序中类 C 的对象 o 所调用的方法 m，o 为一个变量代表某个一个对象，X 是一个形参序列；如果是建立在 API 方法调用上，则安全相关行为不仅要从用户定义的类和方法中抽取，还要从该用户程序所涉及到的所有 API 类和方法来抽取，通常的形式为 $\langle C,o,m\rangle(X)$，代表某个 API 类 C 中方法 m 的调用；如果建立在系统调用一级，则要从用户定义的类和方法、该用户程序所涉及到的所有 API 类和方法以及 Native 方法中抽取，其

形式为 $m(X)$. 安全相关操作集合可以看作是所有安全策略所关心的安全相关方法调用集合,我们用 Σ 来表示.

一个程序的安全相关行为模型应定义该程序所有可能的执行将产生的所有安全相关操作序列. 本文采用参数化扩展上下文无关文法来定义多线程 Java 程序的安全相关行为模型,具体定义如下.

定义 6. 设 P 是一个 Java 多线程程序,参数化扩展上下文无关文法 $G_P=(V_N,V_T,S_0,\alpha,PS)$ 称为 P 的安全相关行为模型,其中

① 非终极符集合 V_N 是方法调用集合,每个非终极符的形式为 $\langle c,m\rangle(X)$,其中 c 是 P 中声明的类名,m 是在类 c 中声明的一个方法名字,X 是方法 m 的所有形参构成的参数元组;

② 终极符集合 V_T 是安全相关系统调用集合以及线程相关的一些操作,即 $V_T=\Sigma\cup\{\langle T,o,\text{start}\rangle,\langle T,o,\text{stop}\rangle,\langle T,o,\text{suspend}\rangle,\langle T,o,\text{resume}\rangle,\langle T,o,\text{join}\rangle \mid T$ 是 P 中声明的线程类,o 是对应的线程对象$\}$;

③ 开始符 S_0 的形式为 $\langle c_0,\text{main}\rangle$,其中程序 P 的 main 方法是在类 c_0 中声明的;

④ 元数函数 α 的定义域是 $V_N\cup V_T$,值域是整数,对于一个符号 $a\in V_N\cup V_T$, $\alpha(a)$ 给出的是对应方法或者系统调用的参数个数;

⑤ PS 是一个有限的产生式集合,每个产生式的形式如下:$\langle c,m\rangle(X)\rightarrow RE(A_1,\cdots,A_n)$,其中:

$\langle c,m\rangle(X)\in V_N$;

X_0 是变量构成的 $\alpha(a_i)$ 元组;

A_i 是可以是形如 $\langle c_i,o_i,m_i\rangle(X_i)$ 的方法调用(其中是 X_i 是类 c_i 的一个对象 o_i 调用其方法 m_i 的所有实参表示),或者是 V_T 中的某个调用(其中参数部分为常量或变量).

每个产生式右部的正则表达式代表的是调用该产生式左部对应的方法而生成的所有可能安全相关操作序列. 下面将给出如何从该方法的源程序自动生成该正则表达式.

本文所采用的基于参数化扩展上下文无关文法的安全相关行为模型的优势在于该模型具有可扩展性、良好的模块性和可复合性,因为建立在低层次的安全相关模型可以通过增加低一层方法得到对应的产生式而扩展,而每个类的每个方法定义对以一个产生式,因此增加新的类定义或者新的程序模块,都不影响原有的模型,只需要增加新增部分的模型,并可与已有模型复合即可.

图 2 给出了一个多线程 Java 程序的例子,该程序实现了简化过的手机助手功能,包括从磁盘读取并初始化通信簿以及从网络下载并存储所需图片两个功能,其中后者使用了线程技术,来防止在单线程下,由于网络不畅引起程序界面或其它功能无法响应的情况出现. 图 3 则给出了图 2 中例子程序所对应的安全相关行为模型(可以通过 3.3 节介绍的方法自动生成).

```
import java.awt.Image;
import java.awt.image.BufferedImage;
import java.io.*;
import java.net.*;
import java.util.StringTokenizer;
import com.sun.image.codec.jpeg.*;

public class PhoneAssistant {
 public static void main(String[] args) throws Exception{
              initLogo();initContacts(); }

 private static void initLogo() {
  ImageDownloader downloader=new ImageDownloader();
  Thread t=new Thread(downloader);
  t.start(); }

 private static void initContacts()throws IOException{
         BufferedReader in=new BufferedReader(new
         FileReader("/home/sensitivefiles/contacts.dat"));
         Contact[] contacts=readData(in);
         in.close();}
         for (Contact e : contacts) System.out.println(e);
         }

static Contact[] readData(BufferedReader in) throws IOException
    {int n=Integer.parseInt(in.readLine());
     Contact[] contacts=new Contact[n];
     for (int i=0; i<n; i++){
         contacts[i]=new Contact();contacts[i].readData(in);}
       return contacts;
    }
}

class Contact{
  public Contact() {}
  public Contact(String n) {name=n;}
  public String getName() {return name;}
  public String toString() {return name; }
  public void readData(BufferedReader in) throws IOException
    {  String s=in.readLine();
       StringTokenizer t=new StringTokenizer(s, "|");
       name=t.nextToken();
    }
  private String name;
}

class ImageDownloader implements Runnable{
  public void run() {
   try {URL url=new URL("http://www.google.com/
                        images/logo_sm.gif");
    Image src=javax.imageio.ImageIO.read(url);
    int wideth=src.getWidth(null);
    int height=src.getHeight(null);
    BufferedImage tag=new BufferedImage(wideth/2, height/2,
    BufferedImage.TYPE_INT_RGB);
    tag.getGraphics().drawImage(src,0,0,wideth/2,height/2,
                                        null);
    FileOutputStream out=
                  new FileOutputStream("/tmp/logo_sm.jpg");
    JPEGImageEncoder encoder=
                  PEGCodec.createJPEGEncoder(out);
    encoder.encode(tag);
    out.close();
   } catch (MalformedURLException e) {e.printStackTrace();}
  }
}
```

图 2　多线程 Java 程序例子

```
〈PhoneAssistant, main(args)〉→
  〈PhoneAssistant, initLogo()〉〈PhoneAssistant, initContacts()〉

〈PhoneAssistant, initLogo()〉→
  〈Thread, t, start()〉

〈PhoneAssistant, initContacts()〉→
  〈PhoneAssistant,, readData(in("/home/svn/sensitivefiles/
                    contacts.dat"))〉
  〈BufferReader, in, close()〉(System.out.println(e))*

〈PhoneAssistant, readData(in)〉→
  〈BufferReader, in, readline()〉〈Integer,, ParseInt(temp)〉
  (〈Contact, contact[i], readData(in)〉)*

〈Contact, Contact(n)〉→
〈Contact, GetName(n)〉→
〈Contact, toString()〉→

〈Contact, readData(in)〉→
  〈BufferedReader, in, readLine〉
  〈StringTokenizer, t, nextToken()〉

〈ImageDownloader, run()〉→
  〈javax.imageio.ImageIO.read(url
        ("http://www.google.com/images/logo_sm.gif"))〉
  〈Image, src, getWidth()〉〈Image, src, getHeight()〉
  〈BufferedImage, new(wideth/2,height/2,
                    BufferedImage.TYPE_INT_RGB)〉
  〈BufferedImage, tag, getGraphics().drawImage(src, 0, 0,
                    wideth/2, height/2, null)〉
```

图 3 多线程 Java 程序例子对应的安全相关行为模型

3.3 安全相关行为模型自动生成

设 P 是一个多线程 Java 程序，则对于 P 中声明的每个类中的每个方法生成一个产生式，具体生成方法如下：

若类 c 中的一个方法 m，其形参为(fp_1，…，fp_n)，其程序体为语句序列{S_1，…，S_m}，则该方法对应的产生式为

〈c，m(fp_1，…，fp_n)〉→GenM【S_1】…GenM【S_m】，

其中 GenM 是一个抽象函数，用于根据具体的语句、表达式或者变量的形式生成相应的正则表达式，其具体定义如下(其中赋值语句、表达式、变量、条件语句、循环语句与文献[7-8]中的生成方法大致相同)：

(1)赋值语句

对于赋值语句，依次生成变量和表达式对应的安全相关操作序列.

GenM【$V=E$】=GenM【V】GenM【E】.

(2)表达式

根据表达式的计算顺序依次分析对应的运算分量，如果是常量、简单变量情形则为空，如果是方法调用则按照下面方法调用的情形生成相应的部分.

GenM【C】=ε，C 代表常量；

GenM【V】如情形(3)，V 代表变量；

GenM【$e_1 \omega\, e_2$】=GenM【e_1】GenM【e_2】，ω 是双目操作符；

GenM【$\omega\, e$】=GenM【e】，ω 是一个单目操作符；

GenM【(e)】=GenM【e】.

(3)变量

对于简单变量情形，生成空串；对于复杂变量情形，如果包含表达式或者方法调用，则根据其成分的计算顺序，依次生成对应的部分，即

GenM【V】=GenM【p_1】…GenM【p_k】，

其中[p_1，…，p_k]表示计算 V 地址过程中顺序用到的方法调用序列.

(4)条件语句

针对 if-then 和 if-then-else 情形条件语句，两个分支通过“|”来表示：

GenM【if(E) S】=GenM【E】(GenM【S】|ε)；

GenM【if(E) S_1 else S_2】=

GenM【E】(GenM【S_1】|GenM【S_2】).

对于 Switch 语句的多分支情形，可以生成由多个“|”来对应不同分支的正则表达式.

GenM【switch (E) {*casebody*}】=

GenM【E】(GenMCase【*casebody*】，

其中

casebody=c_1 : s_1；[break;] …；case c_n : s_n；

[break;]；default : s;[break;].

GenMCase 函数给出的是如何从 case 序列自动生成相应的正则表达式：

①一个 case 分支情形：

GenMCase【case c_1 : s；[break;]】=GenM【S】；

GenMCase【default : s;[break;]】=GenM【S】.

②多个 case 分支，其中第一个分支末尾没有 break 语句：

GenMCase【case c_1 : s_1；othercases】=

GenM【S_1】GenMCase【othercases】.

③多个 case 分支，其中第一个分支末尾有 break 语句：

GenMCase【case c_1 : s_1；break；othercases】=

(GenM【S_1】|GenMCase【othercases】).

(5)循环语句

GenM【while E do S】=

(GenM【E】GenM【S】)* GenM【E】；

GenM【do S while E】=(GenM【S】GenM【E】)+.

对于 For 语句可以根据其语义可以很直接地改写成由赋值语句和条件语句构成的复合语句，进而可以应用前面的方法自动生成对应的正则表达式，这里就不再赘述了.

(6)特殊的线程方法调用

对于线程的一些特殊方法调用将做特殊处理，

这些包括 start,stop,suspend,resume,join.

$\mathrm{GenM}【O.M()】=\langle T_1,O,M\rangle|\cdots|\langle T_n,O,M\rangle$,

其中$\{T_1,\cdots,T_n\}$是 O 对应的所有可能的线程类,可以通过静态分析得到;

(7) 安全相关操作

$$\mathrm{GenM}【A(X)】=A(X).$$

(8) 其它方法调用

$$\begin{aligned}\mathrm{GenM}【O.M(e_1,\cdots,e_n)】=&\mathrm{GenM}【e_1】\cdots\mathrm{GenM}【e_n】\\&(\langle C_1,O,M\rangle(e_1,\cdots,e_n)\\&|\cdots\\&|\langle C_n,O,M\rangle(e_1,\cdots,e_n)),\end{aligned}$$

其中$\{C_1,\cdots,C_n\}$对应的是对象 O 所有可能的类构成的集合.$\{x_1,\cdots,x_n\}$是一个变量集合,其中每个 x_i或者是一个常量,或者是一个变量.

针对方法 m 所生成的产生式,出现在其右部的所有方法调用按出现顺序排列为

$$\langle C_1,O_1,M_1\rangle(e_{1,1},\cdots,e_{1,i1})\cdots\langle C_i,O_k,M_k\rangle(e_{k,1},\cdots,e_{i,ik}).$$

3.4　安全相关行为模型的精化

通过上述方法对多线程 Java 程序分析后,得到初步的安全相关行为模型,接下来需要对该模型进行一些处理,主要是将通过简单控制流分析,可以建立变量(包括方法调用实参)以及方法 m 的形参之间的赋值关系,基于这个可能的赋值关系,将对每个方法调用$\langle C_j,O_j,M_j\rangle(e_{j,1},\cdots,e_{j,ij})$的参数依次进行重名,重名规则如下:

如果 $e_{j,in}$为常量,则不变;

否则,如果 $e_{j,in}$为变量,而且和方法 m 的某个形参有赋值关系,则改名为该形参名;

否则,如果 e_i为变量,而且和$\langle C_j,O_j,M_j\rangle$前面的任何方法调用$\langle C'_j,O'_j,M'_j\rangle$的参数 e'有赋值关系,则改写成 e'对应的变量名;

否则,如果 e_i为变量,而且和$\langle C_j,O_j,M_j\rangle$前面的任何方法调用的参数都没有赋值关系,则改写成一个没有一个不同于产生式中任何其他变量名的一个特殊名字.

所谓赋值关系是指通过控制流分析可以得出这两个变量或者参数在方法 m 执行的某个路径上可能相同.目前没有考虑表达式等价性.

3.5　循环问题处理

基于多线程 Java 程序安全相关行为模型,本文将试图生成执行该程序的所产生的所有可能的安全相关操作序列,进而进一步检查这些安全相关操作序列是否满足安全策略的定义.然而,由于通过静态分析无法确定循环次数,所以在安全相关行为模型把循环处理成了 * 闭包或者+闭包,这样将导致该模型对应的是一个无穷集合,即该模型所接受的一些安全操作序列并不能够通过执行对应程序而得到.本文给出两种简单的解决方法:

(1) 设定固定的最大循环次数 $MaxRepeatNum$,将所有的 * 闭包和+闭包改为 $MaxRepeatNum$ 次重复;

(2) 通过测试计算出每个循环的最大循环次数,进而在模型中把对应的 * 闭包和+闭包替换成测试中得到的最大循环次数;

即要把循环的处理改为,如果确定循环最大次数为 n,则

$$\mathrm{GenM}【\text{while } E \text{ do } S】=(\mathrm{GenM}【E】\mathrm{GenM}【S】)^n\mathrm{GenM}【E】;$$

$$\mathrm{GenM}【\text{do } S \text{ while } E】=(\mathrm{GenM}【S】\mathrm{GenM}【E】)^n.$$

另外,方法间显示或者隐式的递归调用将同样引入无限循环,在确定对应的调用环路后,可以采用上述策略,但是需要在模型检查时针对每个环路引入计数器来处理.

4　安全相关行为模型的静态检查

4.1　安全相关行为模型静态检查的含义

所谓安全相关行为模型检查就是将一个程序的安全相关行为模型和用户的安全策略定义相比较,以确定该安全相关行为模型是否符合安全策略的要求.通常安全策略是给出的不允许的行为,因此一个程序的安全相关行为模型检查的含义就是:对于该程序的所有可能的执行所产生的安全相关操作序列,检查是否包含对应安全策略所定义的某个符号串,如果都不包含则表示满足,否则只要存在一个执行包含就表示不满足.

定义 7.　设有一个安全策略的 $EFSA$ 定义 SP,一个程序 P,若 P 的一次执行所产生的安全相关操作序列为$\alpha=A_1(X_1)\cdots A_n(X_n)$,则

如果存在 α 的某个子串 $\beta=A_i(X_i)\cdots A_{i+m}(X_{i+m})$是 SP 所接受的,而且 $1\leq i\leq n-m$,则称 α 不满足 SP,即 P 的本次执行不满足安全策略 SP;否则,称 α 满足 SP,即 P 的本次执行满足安全策略 SP.

本定义是针对程序的一次执行,说明如果该执行所对应的安全相关行为序列中包含一个子串,该子串被安全策略定义所接受,则可以确定该次执行不满足安全策略;如果所有子串都不被安全策略定义所接受,则本次执行满足安全策略.

定义 8. 设有一个安全策略的 EFSA 定义为 SP 以及一个程序 P 的安全相关行为模型 M，则

如果 $L(M,Y)$ 中至少存在一个符号串 α 不满足 SP，则称 P 不满足安全策略 SP（其中 Y 是对应 M 中开始符号的形参序列）；

否则，称 M 满足安全策略 SP.

本定义说明如果能够找到安全相关行为模型所接受的某个符号串不满足安全策略，则该模型就不满足安全策略. 安全相关行为模型的静态检查将尽力找到所有不满足安全策略的、该模型所接受的安全操作序列.

静态检查安全相关行为模型的主要思想是：根据安全相关模型的定义，逐步生成所有可能的执行，每个执行是由多个线程以及已经生成的安全相关操作序列集合构成的，而每个线程则对应一个正则表达式，代表其剩余的可能的安全操作序列集合. 此外，每个安全相关操作序列还对应着安全策略定义中的状态环境对的集合. 每推导出一个安全相关操作，则根据安全策略定义来更新状态环境集合，如果其中包含一个终止状态，则表示不满足，否则继续分析，直到发现所有执行的所有线程或者其生成的安全相关操作序列不满足安全策略，或者其剩余正则表达式为空（表示执行完）.

4.2 抽象域和预定义函数

对于一个多线程 Java 程序 P，定义如下抽象域：

① 安全相关操作集合，用 Σ 表示；

② 定义在 Σ 上的所有安全策略集合，用 SP 表示，定义在 Σ 上的一个安全策略可以用 s 表示，$s=(Q,q_0,F,V,Env,\Sigma,\delta)\in SP$；

③ 安全相关行为模型集合，用 $Models$ 表示，P 的安全相关行为模型 $M=(V_N,V_T,S_0,\alpha,PS)\in Models$；

④ 安全相关行为模型中产生式右部的正则表达式集合，用 RE 表示；

⑤ 线程动态环境：$\delta\in ThreadDEnv=INT\rightarrow RE$；

⑥ 线程静态环境：$\sigma\in ThreadSEnv=TID\rightarrow INT$；

⑦ 线程环境：$\tau\in ThreadEnv=ThreadDEnv\times ThreadSEnv$；

⑧ 安全操作序列：$trace\in Trace=\Sigma^*$；

⑨ 安全操作序列集合：$traces\in TraceS=2^{Trace}$；

⑩ 状态集合：$qs\in SS=2^{(Q\times Env)}$；

⑪ 执行：$Exec=ThreadEnv\times 2^{(Trace\times SS\times BOOL)}$；

⑫ 执行集合：$ExecS=2^{Exec}$.

基于这些抽象域，将定义如下预定义函数：

(1) UnsatisTermi：$ExecS\rightarrow BOOL$.

具体功能是如果每个执行所有安全操作序列对应的标志为假，则返回真，否则返回假.

具体定义如下：

UnsatisTermi ($execs$)=
if $execs=\varnothing$ then true
else let $exec$=GetOneExec($execs$) in
let $(\tau,ts)=exec$ in
($ts=\varnothing\rightarrow$UnsatisTermi ($execs$-{$exec$}),
let (t,qs,b)=GetOneTrace(ts) in
($b\rightarrow$false, UnsatisTermi (($execs$-{$exec$})$\cup$
(τ, $tl(ts)$))))).

(2) SatisTermi：$ExecS\rightarrow BOOL$.

具体功能是如果每个执行的线程环境为空，且没有一个安全操作序列对应的标志为假，则返回真，否则为假.

具体定义如下：

SatisTermi ($execs$)=
if $execs=\varnothing$ then true
else let $exec$=GetOneExec($execs$) in
let (τ, ts)=$exec$ in
($ts=\varnothing\rightarrow$SatisTermi($execs$-{$exec$}),
let (t,qs,b)=GetOneTrace(ts) in
(($\exists id\in domain(\tau),\tau(id)\neq\varepsilon)\vee\neg b\rightarrow$false,
SatisTermi (($execs$-{$exec$})$\cup(\tau,tl(ts))$))))).

(3) AddOne：$2^{(Trace\times SS\times BOOL)}\times\Sigma\times SP\rightarrow 2^{(Trace\times SS\times BOOL)}$.

具体功能是：如果当前序列已经不满足安全策略，则什么都不做，否则，针对每一个安全相关操作序列及其状态环境，把新的安全相关操作加入当前已生成的所有操作序列后面，并更新对应的状态环境，如果其中一个状态属于终止状态则修改该序列标志位为假. Trans 函数就是根据当前的安全策略定义，当前的安全相关操作以及当前的状态环境集合 qs，返回转向新的状态环境集合.

具体定义如下：

AddOne(ts,$a(X)$,p)=
if $ts=\varnothing$ then $\varnothing$
else let (t,qs,b)=GetOneTrace(ts) in
let ts'=AddOne(ts-{(t,qs,b)},$a(X)$,p) (ts) in
($b\rightarrow$let qs'=trans(p,$a(X)$,qs) in
(($\exists q,q\in qs'\wedge q\in p.F)\rightarrow$
{(::$a(X)$,qs',false)}$\cup ts'$,
{(t::$a(X)$,qs',true)}$\cup ts'$)),
(t,qs,b)$\cup ts'$).

(4) RS：$2^{TID}\times TID\times Exec\rightarrow ExecS$.

具体功能是将在执行 $exec$ 中第一个操作为 resume的线程集合 TS 与第一个操作为 suspend 的线程 t 对应起来,得到新的执行集合.

具体定义如下:

RS (TS, t, $exec$)=
 if $TS=\varnothing$ then $\varnothing$
 else let t'=GetOneTid(TS) in
 let $execs$=RS(TS-{t'},t,$exec$) in
 let (τ,ts)=$exec$ in
 let (δ,σ)=τ in
 let A: $re=\delta(t)$ in
 let B: $re'=\delta(t')$ in
 let $exec'=(\delta[re/t][re'/t'],\sigma)$ in
  $execs\cup exec'$

4.3　安全相关行为模型静态检查的抽象描述

本文基于多线程 Java 程序的安全相关行为模型,通过一个抽象函数定义给出了如何检查该模型是否满足安全策略定义的过程.

(1)安全相关行为模型检查函数及其解释

函数空间为 $SecurityChecker\in Model\times SP\rightarrow BOOL\times Traces$.

具体功能:对于给定的一个安全相关行为模型和一个安全策略,如果该安全相关行为模型满足该安全策略,则返回 true 以及生成的所有安全相关操作序列,如果该模型不满足该安全策略,返回 false 以及所找到的那些违反安全策略的操作序列.

初始执行情形:执行集合中只有一个主线程,该线程的剩余正则表达为 main 方法对应的正则表达式,当前生成的安全操作序列集合为空.

结束执行情形:

(i)只找一种不满足情形:有一个执行的一个安全操作序列对应标志为假,则结束(过于简单);

(ii)尽可能找到更多不满足情形:每个执行所有安全操作序列对应的标志为假,说明找到不满足安全策略的情形,用 UnsatisTermi($execs$)表示;

(iii)满足情形:每个执行的线程环境为空,且没有一个安全操作序列对应的标志为假,说明没有找到不满足安全策略的情形,用 SatisTermi($execs$)表示.

函数具体定义如下:

SecurityChecker (m,p)=
let τ=([$mid\rightarrow$Right(m.PS⟨c,main⟩)], [$T_{main}\rightarrow mid$])
 in let $execs$=AbsInter(τ,{}),m,p)
  in (UnsatisTermi($execs$)→(false, GetTraces($execs$)),
   SatisTermi($execs$)→(true,GetTraces($execs$)))

其中 GetTraces($execs$)是从执行集合中获得所有的安全操作序列.

(2)一步处理函数

函数空间为 $AbstInter\in ExecS\times Model\times SP\rightarrow ExecS$.

具体功能:给定一个执行集合以及安全相关行为模型和安全策略,返回经过一步执行后得到的新的可能执行集合.其主要思想:选择一个执行,对于其所有线程,根据其剩余正则表达式中的第一个符号形式,做相应的分析和检查动作,具体定义如下:

AbstInter($execs$,m,p)=
 while¬(UnsatisTermi($execs$) ∨ SatisTermi($execs$)) do
 {let $exec$=GetOneExec($execs$)
  in let $\mu_0=\lambda i.$ false
   in let $execs'$=DealOneExec($exec$,m,p,μ_0)
    in AbstInter((($execs$-{$exec$}))$\cup execs'$, m, p)
 }

其中 GetOneExec($execs$)是指从执行集合 $execs$ 中任取一个执行返回.

(3)一个执行中所有线程的处理

函数空间为 $DealOneExec\in Exec\times Model\times SP\times Mark\rightarrow ExecS$.

$Mark=TID\rightarrow BOOL$.

具体功能:处理该执行中所有线程,得到它们顺序交叉执行的所有可能执行集合.

具体定义如下:

DealOneExec($exec$, m, p, μ)=
 let (τ,ts)=$exec$ in
 let (δ,σ)=τ in
 forany (tid| $tid\in domain(\delta)\wedge(\neg\mu(tid))$)do
 { let $newexec$=DealOneThread(tid, $exec$,m,p) in
  let $execs$=DealOneExec($exec$,m,p,$mark[true/tid]$)
   in $execs\cup exec$
 }

(4)每个线程的处理

函数空间为 $DealOneExec\in TID\times Exec\times Model\times SP\rightarrow ExecS$.

具体功能:针对一个执行中的某个线程,将根据该线程剩余正则表达式中第一个操作符号来得到若先执行该操作后的新执行,具体如下:

如果是带括号情形(t),则去掉括号;

如果是选择情形 $t1|t2$,则把每个分支作为一个执行,从而获得两个执行;

如果是循环情形 t^n,依次处理,每个循环中新生

成线程或者变量都增加循环次数 n 处理；

如果是顺序情形，则取出第一个符号，根据该符号的情形来处理：

① 如果线程 tid 剩余正则表达式的第一个符号是安全相关操作，则把该操作加入当前已生成的所有操作序列后面，更新对应的状态（由函数 $AddOne(ts,a(X),p)$ 完成），并得到新的一个可能执行：其中线程 tid 的剩余正则表达式去掉本操作；

② 如果线程的第一个符号是调用开始线程方法 start，其中 T 是一个线程类，则建立一个新的 thread 编号，修改对应线程环境，包括该线程的执行体部分为对应线程类中 run 方法的产生式右部；

③ 如果线程的第一个符号是终止线程的方法 stop，其中 T 是一个线程类，则把该线程剩余正则表达式变为空；

④ 如果线程的第一个符号是悬挂线程的方法 suspend，则若在当前执行中没有一个线程的第一个操作是对 t 进行 resume，则什么都不做；否则，找出所有以第一个操作室对 t 进行 resume 的线程（由函数 $FindResumeT$ 完成），一一对应起来得到新的执行集合（由函数 RS 完成）；

⑤ 如果线程的第一个符号是恢复线程的方法 resume，则若在当前执行中没有一个线程的第一个操作是对 t 进行 suspend，则什么都不做；否则，找出所有以第一个操作室对 t 进行 suspend 的线程，一一对应起来得到新的执行. 不再重复；

⑥ 如果线程的第一个符号是线程加入方法 join，则根据该操作含义，将等到 t 结束后再执行，这样就需要判定线程 t 的剩余正则表达式是否为空，如果不为空则不变，如果为空则处理当前线程的下一个操作；

⑦ 如果线程的第一个符号是普通方法调用，则需要用该方法调用对应的正则表达式（要经过参数更新）替换第一个符号即可，其它不变.

具体定义如下：

DealOneThread(tid, $exec$, m, p)=

{

let (τ, ts)=$exec$ in

let (δ,σ)=τ in

case $\delta(tid)$ of { /* 看正则表达式的形式 */

(t):{(($\delta[t/tid]$,σ),ts)}

$t1|t2$:{(($\delta[t1/tid]$,σ),ts), (($\delta[t2/tid]$,σ),ts)}

t^n: {if n=0 then $\varnothing$ else {(($\delta[t\,t^{n-1}/tid]$,σ),ts)}}

A: re :

case A of {

A=$a(X)\in\Sigma$(安全相关操作)：

{let ts'=AddOne(ts,$a(X)$, p) in

let $exec'$=(($\delta[re/tid]$, σ),ts')

in {$exec'$}

}

A=⟨T,t,start⟩() (线程执行)：

{let tid'=NewTID() in

let re'=Right(M. PS(⟨T, run⟩) in

let τ'=(($\delta[re'/tid'][re/tid]$, $\sigma[tid'/t]$)

let in $exec'$=(τ',ts) in {$exec'$}

}

A=⟨T,t,stop⟩() (线程停止)：

{let tid'=$\sigma(t)$

let in $exec'$=(($\delta[\varepsilon/tid'][re/tid]$, $\sigma[\perp/t]$),ts)

in {$exec'$}

}

A=⟨T,t,suspend⟩() (线程悬挂)：

{let TS=FindResumeT(δ) in

if $TS\neq\varnothing$ then RS(TS, t, $exec$)

else $\varnothing$

}

A=⟨T,t,resume⟩()(线程恢复)：

{let TS=FindSuspendT(δ) in

if $TS\neq\varnothing$ then RS(TS, t, $exec$)

else $\varnothing$

}

A=⟨T,t,join⟩(t') (线程加入)：

{($\delta(\sigma(t))=\varepsilon\rightarrow$

{(($\delta[re/tid]$,σ),ts)},

{$exec$})

}

default(普通方法调用)：

{let ⟨C,o,m⟩(X)=A in

let $re(Y)$=Right(M. PS(⟨C,m⟩(X)))

in $exec'$=(($\delta[re[X/Y::\delta(tid)/tid]$,$\sigma$),$ts$)

}

}}

4.4 安全相关行为模型静态检查例子说明

下面将给出了针对图 3 中的安全相关行为模型和图 1 中的安全策略完成的静态检查过程说明：

安全相关操作集合包括 javax. imageio. ImageIO.read(当其参数为非本地地址是就属于 socket

类安全相关操作)和 Bufferer.readLine(属于 FileRead 类安全相关操作).

初始情形下,只有一个执行,其中线程环境为

$\tau=([1\rightarrow\langle$PhoneAssistant, initLogo()$\rangle$
$\langle$PhoneAssistant, initContacts()$\rangle]$,
$[T_{main}\rightarrow 1])$.

已经生成的安全相关操作序列、状态和是否满足安全策略标志位构成的三元组集合为空:{}.

接下来,因为该执行只有一个线程,因此处理该线程 T_{main} 的剩余正则表达式部分的第一个符号$\langle$PhoneAssistant, initLogo()$\rangle$,是一个普通方法调用,故通过替换后得到新的线程环境为

$\tau=([1\rightarrow\langle$Thread,$t$,start()$\rangle$
$\langle$PhoneAssistant,initContacts()$\rangle]$,
$[T_{main}\rightarrow 1])$.

再接下来,处理$\langle$Thread,t,start()$\rangle$,建立一个新的线程编号及其对应的环境,得到

$\tau=([1\rightarrow\langle$PhoneAssistant, initContacts()$\rangle$,
$2\rightarrow\langle$javax.imageio.ImageIO.read(url(
"http://www.google.com/images/logo_sm.gif"))$\rangle$
$\langle$Image, src, getWidth()$\rangle$
$\langle$Image, src, getHeight()$\rangle$
$\langle$BufferedImage,,new(wideth/2,height/2,
BufferedImage.TYPE_INT_RGB)$\rangle$
$\langle$BufferedImage,tag,getGraphics().
drawImage(src,0,0,wideth/2,height/2,null)$\rangle]$,
$[T_{main}\rightarrow 1,T_{downloader}\rightarrow 2])$

此时,已经有两个线程,经过一步处理,将产生两个可能的执行,如图 4.

(1) $\tau=([1\rightarrow\langle$PhoneAssistant,,
readData(in("/home/svn/sensitivefiles/contacts.dat"))$\rangle$
$\langle$BufferReader, in, close()$\rangle$(System.out.println(e))*,
$2\rightarrow\langle$javax.imageio.ImageIO.read(url
("http://www.google.com/images/logo_sm.gif"))$\rangle$
$\langle$Image, src, getWidth()$\rangle\langle$Image, src, getHeight()$\rangle$
$\langle$BufferedImage,,new(wideth/2,height/2,
BufferedImage.TYPE_INT_RGB)$\rangle$
$\langle$BufferedImage,tag,getGraphics().drawImage(src,
0,0,wideth/2,height/2, null)$\rangle]$,
$[T_{main}\rightarrow 1,T_{downloader}\rightarrow 2])$

(2) $\tau=([1\rightarrow\langle$PhoneAssistant, initContacts()$\rangle$,
$2\rightarrow\langle$Image, src, getWidth()$\rangle\langle$Image, src, getHeight()$\rangle$
$\langle$BufferedImage,,new(wideth/2,height/2,
BufferedImage.TYPE_INT_RGB)$\rangle$
$\langle$BufferedImage,tag,getGraphics().drawImage(src,
0,0,wideth/2,height/2, null)$\rangle$
$[T_{main}\rightarrow 1,T$ downloader$\rightarrow 2])$

安全相关操作序列为
$\langle$javax.imageio.ImageIO.read(url("http://www.google.com/images/logo_sm.gif"))$\rangle$

图 4　执行集合例子

对其中第一个执行,经过两步处理后将包含一个违反安全策略的执行,如图 5.

$\tau=([1\rightarrow\langle$Integer,,ParseInt(temp)$\rangle$
$\langle$Contact, contact[i], readData(in))$\rangle$*$\langle$BufferReader, in, close()$\rangle$(System.out.println(e))*,
$2\rightarrow\langle$Image, src, getWidth()$\rangle\langle$Image, src, getHeight()$\rangle$
$\langle$BufferedImage,,new(wideth/2,height/2,
BufferedImage.TYPE_INT_RGB)$\rangle$
$\langle$BufferedImage,tag,getGraphics().drawImage(src,
0,0,wideth/2,height/2,null)$\rangle]$,
$[T_{main}\rightarrow 1,T_{downloader}\rightarrow 2])$

安全相关操作序列为
$\langle$PhoneAssistant,,readData(in("/home/svn/sensitivefiles/contacts.dat"))$\rangle$
$\langle$javax.imageio.ImageIO.read(url
("http://www.google.com/images/logo_sm.gif"))$\rangle$

状态集合为$\{S_3\}$
是否满足安全策略标志:FALSE

图 5　一个违反安全策略的执行例子

可以看出,图 3 的模型违反了图 2 的安全策略,这是因为存在这种情形:在主线程先执行 BufferedReader.readline 来读关键目录"/home/sensitivefiles/"下的文件 contacts.dat,接下来线程 downloader 再执行非本地网络操作,即调用 javax.imageio.ImageIO.read(url),其中 url 对应的链接地址是 http://www.google.com/images/logo_sm.gif.

4.5　效果分析

静态属性检查方法通常是合理的,但很难具有完全性,如果二者都不具备则是试探性[2].本文中生成的安全相关行为模型是完全的,但是为了保证安全性检查的终止性,因此限制了循环(递归)的次数,从而导致本文的安全相关行为模型静态检查方法是一种试探性方法.但是本方法具有可扩展性和良好的模块性,可以自动检查安全性,因此可以帮助提供有用的信息.

为了验证所提出方法的有效性,本文针对若干个典型的文件访问控制安全策略,设计了一些多线程 Java 程序例子,开展了安全相关行为模型自动生成和静态检查的实验.实验表明,本文方法极大地依赖于所建立的参数之间关系和安全策略定义的复杂度.具体如下:

(1)针对系统调用一级的安全策略,设计了特定的违反安全策略的例子程序,通过静态检查都给出了不满足安全策略的结果,而采用运行时监控则是在部分情形下发现违反安全策略的情形,这是由多线程程序的不确定性造成的;

(2)针对应用程序一级的安全策略,如果安全策略中包含复杂的条件,特别是要求依据程序中特定变量来判定,而且没有更多的参数间赋值关系,则即便有违反安全策略情形也不能检查出来.因此,如

果结合更精确的数据流分析[9]和控制流分析[3]来精化分析结果，如：通过可到达性分析可以删掉一些不可能出现的安全操作序列，采用别名分析可以提供更多可能的参数赋值关系等，那么安全模型检查结果也将会更精确有效.

此外，由于本文给出的方法需要从源程序中生成的安全相关行为模型，并依据所定义的安全策略来进行检查，而类似于文献[5]中应用程序的源代码无法获得，因此针对更大规模的多线程 Java 移动代码的实验和比较尚未开展.

5 在 MCC 方法中应用

携带模型代码(Model-Carrying Code，MCC)方法[5]是由 Stony Brook 大学的 Sekar 等提出的，是一个比较全面地解决非信任移动代码在主机上安全执行问题的框架. 其主要思想是：由移动代码的生产方提供代码安全相关行为信息(模型)，移动代码的使用方可以形式化地对该信息进行推理，检查是否与自己的安全策略相符合，如果符合，则执行该代码；如果有冲突，则代码的使用方被允许重新修订其安全策略. 特别地，还支持运行时对安全策略的强制实施，来确保该执行不会造成破坏[6]. 作者及其所在研究小组在 Java 平台上初步实现了 MCC 方法[6,8]，其框架如图 6 所示，其中安全相关行为模型生成器(model generator)和静态检查器(static checker)是 MCC 方法中的重要组成部分. 在文献[8]中完成了顺序 Java 程序安全相关行为模型自动生成，而本文实现了多线程 Java 程序的安全相关行为的自动生成，二者统一采用参数化扩展上下文无关文法作为模型的定义形式，从而确保可以对 Java 顺序程序和部分多线程程序的安全相关行为模型的静态检查.

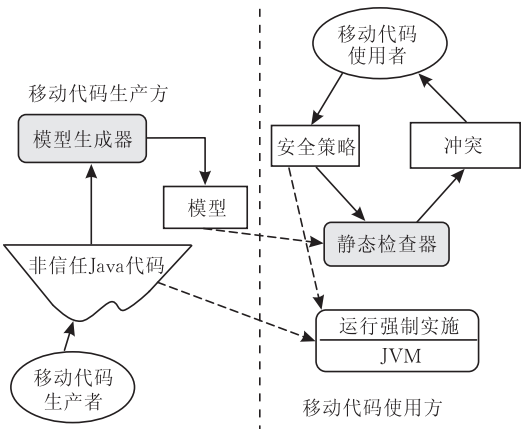

图 6 携带模型代码方法在 Java 平台实现

每个 Java 程序的安全相关行为模型被作为. class 文件的一部分传递到移动代码的使用方.

6 相关工作比较

关于软件安全性分析是可以从软件系统设计、实现、测试等各个方面来开展的[1-3]，其中基于语言的安全性分析结合了程序设计语言以及软件安全性研究，通过模型检查、静态分析、类型系统、形式逻辑分析等为安全策略定义、信息流和控制流分析以及安全漏洞检查等提供了方法和技术支持[1]. 本文采用了静态分析的方法，在解决循环问题时可以采用静态固定循环次数或者采用动态方法(测试)来提供有用的信息，面向的问题是检查安全相关行为.

近年来，关于多线程程序分析研究比较活跃，包括采用静态分析、动态分析或者二者结合的方法，大部分集中在研究死锁检测方法和解决数据竞争问题[3-4,9-11]，开展安全行为模型建立和检查的较少[5,8].

安全相关行为模型定义主要采用有限自动机，下推自动机、扩展有限状态机等[5]，然而这些方法不具有模块化性质，不易扩展. 文献[12]中针对类的协议属性给出了静态验证方法. 在文献[7-8]中应用扩展上下文无关文法或者参数化扩展上下文无关文法针对顺序程序给出了安全相关行为或者安全契约的定义，并完成了安全相关行为模型静态检查和动态监控. 本文在这些工作基础上，针对多线程 Java 程序给出安全相关行为模型的定义及其静态检查方法. 基于参数化扩展上下文无关文法的安全相关行为模型是模块化的、可复合的、较少依赖于具体的语言，且较方便于进行安全相关模型检查.

文献[5]用 C 语言建立了安全性相关行为模型，该模型是通过测试时运行监控得到的，而且模型检查没有考虑参数之间的关系. 本文面向的是 Java 多线程程序，而且还考虑到了参数之间的赋值关系，因此解决的问题更复杂.

7 总结和今后工作展望

本文给出了多线程 Java 程序安全相关行为模型的自动生成和检查方法，并应用于 MCC 方法在 Java 平台的实现中，对生成 Java 程序安全相关行为模型和通过静态分析完成安全策略检查提供了较完整的支持. 本文的主要贡献：(1)不同于以往主要针对死锁和数据竞争等问题的多线程程序安全性分析

和检查,本文的核心是将静态分析应用于多线程程序的安全相关行为模型检查;(2)采用了参数化扩展上下文无关文法来表示安全相关行为模型,具有良好的扩展性,而且比文献[5]通过运行监控得到的安全相关行为模型更完全和更精练;(3)应用到 MCC 框架的实现中,为安全执行非信任多线程 Java 程序提供了支持.但是本文工作也存在如下一些问题:(1)仅处理了线程间的顺序行为交织情形,没有考虑 Java Memory Model 情形[9,11],即没有考虑对 Java 线程操作的重排序情况,不允许单纯的线程操作顺序交织执行之外的更多行为的检查;(2)参数间关系建立只处理了简单情形;(3)实验比较简单,没有实现方法间递归调用情形,安全策略只考虑较简单情形.

今后,将在本文方法应用于大规模多线程 Java 程序过程中,进行更广泛的实例分析和效率分析;并结合同步机制处理方法扩展本文方法以包含对同步机制的处理,从而可以减少更多不可能的违反安全策略的情形,提高分析精度;此外,借鉴文献[6-7]中方法,将本文方法和运行监控相结合,对于静态分析不到的情形开始进行运行监控,检查是否满足安全策略,这样会极大地提高效率.

参　考　文　献

[1] Pistoia M, Erlingsson U. Programming languages and program analysis for security: A three-year retrospective. ACM SIGPLAN Notices, 2008, 43(12): 32-39

[2] Rajamani S K. Automatic property checking for software: Past, present and future//Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2006. Pune, India, 2006: 18-20

[3] Jachson D, Rinard M. Software analysis: A roadmap//Proceedings of the Conference on the Future of Software Engineering. Limerick, Ireland, 2000: 133- 145

[4] Artho C, Biere A. Combined static and dynamic analysis. Department of Computer Science, Zürich ETH: Technical Report 466, 2005

[5] Sekar R, Venkatakrishnan V N, Basu S, Bhatkar S, DuVarney D. Model-carrying code: A practical approach for safe execution of untrusted applications//Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'03). Bolton Landing, NY, 2003: 15-28

[6] Wei Da, Jin Ying, Zhang Jing, Zheng Xiao-Juan, Li Zhuo. Open source JVM based implementation of security policy enforcement. Chinese Journal of Electronics, 2008, 37(12A): 35-41(in Chinese)
(魏达,金英,张晶,郑晓娟,李卓. 基于开源 JVM 的安全策略强制实施. 电子学报, 2008, 37(12A): 35-41)

[7] Jin Ying, Li Ze-Peng, Wei Da, Liu Lei. Automatic generation and enforcement of security contract for pervasive application//Proceeding of the TSP2008. Shanghai, China, 2008: 55-60

[8] Li Ze-Peng. Static analysis based atuomatic generation of Java program security behaviour model[M. S. dissertation]. Jilin University, Changchun, 2008(in Chinese)
(李泽鹏. 基于静态分析自动生成 Java 程序安全行为模型[硕士学位论文]. 吉林大学, 长春, 2008)

[9] Naumovich G, Avrunin G S, Clarke L A. Data flow analysis for checking properties of concurrent Java programs//Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering. New Orleans, Louisiana, United States, 1994: 62-75

[10] Demartini C, Sisto R. Static analysis of Java multithreaded and distributed application//Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems. Kyoto, Japan, 1998: 215-222

[11] Ferrara P. Static analysis via abstract interpreation of the happens-before memeory model//Proceedings of the TAP2008. Prato, Italy, 2008: 116-133

[12] Jin Ying. Formal verification of protocol properties of sequential Java programs//Proceedings of the COMPSAC2007 (Vol. 1). Beijing, China, 2007: 475-482

JIN Ying, born in 1971, Ph. D., associate professor. Her research interests include software engineering, mobile code security, formal method.

LI Ze-Peng, born in 1984, Ph. D. candidate. His research interests include mobile code security and software engineering.

ZHANG Jing, born in 1975, Ph. D., lecturer. Her research interests include software formalization and program analysis.

LIU Lei, born in 1960, professor, Ph. D. supervisor. His research interests include program analysis, semantic Web and software formalization.

Background

This work is supported by China National Science Foundation (Grant No. 60603031).

The project aims at solving some key problems in Model Carrying Code (MCC) proposed by Prof. Sekar at Stony Brook University. MCC is a new practical approach to safe execution of untrusted mobile code, where both mobile code producer and consumer have been taken into account. In previous work MCC has been implemented for C programming language and extended FSA has been used both for defining security related behavior model and security policies. In the authors' project they put efforts in implementing MCC in Java platform, which includes (1) searching for a new way to specify security related behavior model; (2) introducing a way to automatically generating security related behavior model from Java programs; (3) providing security policy specification and management; (4) statically checking security related behavior model with respect to security policies; (5) implementing JVM based security policy enforcement. Aiming at these five objectives a complete solution has been introduced, where parameterized extended context free grammar is proposed to represent security related behavior model, which is easy to extend and compose. Based on the methods given the prototype has been developed, and some experiments have been done.

In this paper, the author has taken multithreaded mechanism of Java programs into consideration. An approach is proposed for automatic generation of security related behavior model from multithreaded Java program and static checking of the model generated with respect to security policy defined as extended finite automata. This work extended the authors' previous work by supporting safe execution of multithreaded Java mobile code.