自动推断的搜索模式的油漆类型漏洞

Fabian Yamaguchi，Alwin Maier，Hugo Gascon和Konrad Rieck
戈大学¨德国丁根

抽象风格的漏洞是软件开发中一个持续存在的问题，正如最近发现的“心痛”漏洞惊人地说明的那样。 在这类漏洞中，攻击者控制的数据被从输入源传递到敏感的接收器。 虽然可以自动检测到此漏洞类的简单实例，但涉及多个函数或特定于项目的API的数据流的更微妙的缺陷主要是通过人工审计发现的。通过寻找典型的易受攻击代码模式，提出了不同的技术来加速这一过程。 然而，所有这些方法都需要安全专家在实践中手动建模和指定适当的模式。

在本文中，我们提出了一种自动推断C代码中油漆样式漏洞的搜索模式的方法。 给定一个安全敏感的接收器，例如内存函数，我们的方法自动识别相应的源接收器系统，并构造模式来建模这些系统中的数据流和消毒。 推断的模式表示为代码属性图中的遍历，并能够有效地搜索未排序的数据流-跨越几个函数以及特定于项目的API。 我们在5个开源项目的不同实验中证明了这种方法的有效性。 推断的搜索模式将检查查找已知漏洞的代码量减少了94.9%，并使我们能够发现8个以前未知的漏洞。

索引术语-脆弱性；聚类；图形数据库；

I. 导言

发现和消除软件中的漏洞是计算机安全的一个根本问题。 不幸的是，即使是细微的缺陷，如一次丢失的授权检查或对数据的处理略有不足，也可能导致软件中严重的安全漏洞。 开发更有效的方法来发现这些漏洞的必要性已经变得非常明显，因为密码库开放SSL[1]中最近的“心痛”漏洞和GNU Bash[2]中的“Shellshock”漏洞。 随着程序不断被修改，平台的特性也在不断变化，新的漏洞也会不断出现。 实际上，漏洞发现已经成为一个持续的过程，需要对软件及其安全所依赖的所有技术有深刻理解的专家。

由于脆弱的编程实践的多样性，安全研究主要集中在检测特定类型的漏洞上。 例如，模糊测试[例如20、53]和符号执行[例如49、59]已成功地应用于查找内存损坏漏洞，例如缓冲区溢出、整数溢出和格式字符串漏洞。 符合本研究，方法多样

为了检测Web应用程序漏洞，已经提出了一些建议，例如SQL注入缺陷[例如，10，26]、跨站点脚本[例如，31，48]和缺少授权检查[19,51]。 最近，一些研究人员认识到，系统软件和Web应用程序中的许多常见漏洞都有一个植根于信息流分析的基本主题：数据从攻击者控制的输入源传播到敏感的接收器，而不进行优先消毒，这是一类被称为污染式漏洞的漏洞[见9、10、26、63]。

已经设计出了不同的方法，可以使用描述语言来挖掘污染类型的漏洞，这些语言允许[30,35,63]精确编码危险的编程模式。 从理论上讲，这一想法为构建一个大型的模式数据库提供了可能性，这些模式可以很容易地与源代码相匹配。 不幸的是，与基于签名的入侵检测系统类似，构建有效的漏洞搜索模式需要安全专家投入大量的人工工作。 从安全敏感的接收器开始，专家需要识别相关的输入源、数据流和相应的消毒检查，这往往涉及对项目特定功能和接口的深刻理解。

本文提出了一种从C源代码中自动推断漆样式漏洞搜索模式的方法。 给定一个敏感的接收器，如内存或网络功能，我们的方法自动识别代码库中相应的源接收器系统，分析这些系统中的数据流，并生成反映污染式漏洞特征的搜索模式。 为此，我们将静态程序分析和无监督机器学习的技术结合起来，使我们能够构建通常通过人工分析识别的模式，并允许精确定位不足的消毒，即使数据流跨越几个函数边界并涉及特定于项目的API。 分析人员可以使用此方法为已知通常与漏洞相关的API函数生成模式，并在整个代码库中找到相同漏洞的实例。

我们通过扩展分析平台Joern来实现我们的方法[1] 支持过程分析和发展中的用于提取和匹配搜索模式的插件，那个是对语法、控制流和数据的健壮描述流动这就是脆弱性的特征。 这个平台是建立在基础上的

[1]一个健壮的C/C++代码分析平台，http://mlsec.org/joern

DOI10.1109/sp.2015.54

2015年©，山口法比安。 在IEEE的许可下。
DOI10.1109/sp.2015.54

在高效图形数据库的顶部，并使用所谓的代码属性图[63]来表示源代码的语法、控制流和数据流。 为了利用这种表示，我们将推断的搜索模式表示为图形遍历，这使我们能够快速通过图形，从而在几分钟内扫描大型软件项目以发现潜在的漏洞。

我们用5个开源项目对我们的方法生成已知漏洞的搜索模式的能力进行了实证评估，表明要审查的代码量可以减少94.9%。 此外，我们通过只使用少数生成的搜索模式来发现8个以前未知的漏洞来证明我们的方法的实际优点。

总之，我们的贡献如下。

- *代码属性图的扩展。* 我们扩展了最近提出的代码属性图[63]，以包括有关语句优先级的信息，并启用使用图形数据库查询的过程间分析。
- *调用模式的提取。* 我们提出了一种利用聚类算法从源代码中提取调用模式的新方法，包括参数的定义和它们所经历的消毒。
- *搜索模式的自动推断。* 最后，我们展示了调用模式如何能够以图形遍历的形式转换为搜索模式，从而能够对大型代码库进行审计。

本文的其余部分组织如下：在第二节中，我们介绍了油漆式漏洞和代码属性图的基本知识。 然后，我们在第三节中介绍了代码属性图的扩展。 我们的搜索模式自动推理方法在第四节中给出，并在第五节中进行了评估。我们分别在第六节和第七节中讨论了我们的方法和相关工作的局限性。 第八节结束文件。

II. 背景

漏洞发现是计算机安全的一个经典话题，因此，人们提出了许多方法，重点是各种类型的漏洞和技术。 在本节中，我们简要回顾了与我们的方法相关的方法。 我们首先在第二-A节中讨论了油漆式漏洞的概念，因为这些是我们在整个论文中处理的漏洞类型。 我们继续描述如何使用第二-B节中的代码属性图来发现这些类型的漏洞，这是一种为基于模式的漏洞发现而设计的表示，我们将其扩展到过程间分析中。

A. 污点漏洞

taint风格的漏洞一词源于taint分析，这是一种通过程序跟踪数据传播的技术。 污染分析的一个目标是识别从攻击者控制的源到不经过消毒的安全敏感汇的数据流。这一程序要求定义(a)适当的来源，(b)相应的来源

```
/* SSL/d1_both*/
// [...]
国际dtls1_process_heartbeat(SSL*s)
{
  无符号字符*p=&s->s3->rrec.data[0], *无符号短hbtype;
  无符号int有效载荷;
  无符号int填充=16; /* 使用最小填充*/
  /* 先读取类型和有效载荷长度*hbtype=*p++;
  N2S(p, 有效载荷);
  如果(1+2+有效载荷+16>s->s3-rrec.length)返回0; /* 每RFC6520秒默
    默丢弃。 4*/
  pl=p;
  // [...]
  如果(hbtype==TLS1_HB_REQUEST){无符号
   字符*缓冲区, *血压; INTR;
   // [...]
   缓冲区=OPENSSL_malloc（1+2+有效负载+填充）; BP=缓冲区;
   /* 输入响应类型、长度和复制有效载荷*/
   *英国石油++=TLS1_HB_RESPONSE;
   s2n(有效载荷, bp); memcpy(bp,
   pl, 有效载荷); bp+=有效载荷;
   /* 随机填充*/RAND_pseudo_bytes(bp, padding);
   r=dtls1_write_bytes, TLS1_RT_HEARTBEAT, 缓冲区,
                                3+有效载荷+填充);
   // [...]
   如果(r<0)返回r;
  }
  // [...]
  返回0;
}
```

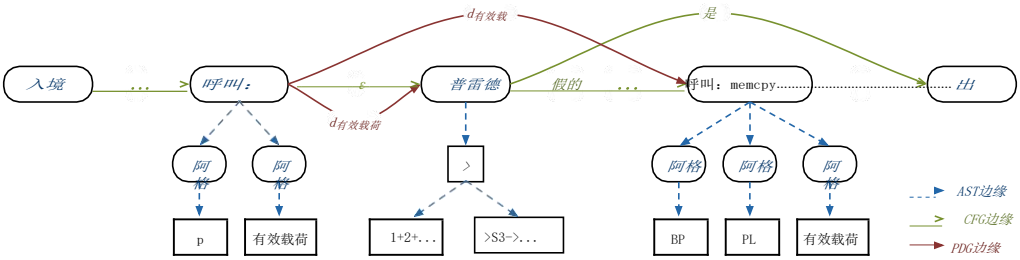
无花果。 1：开放SSL中的“心痛”漏洞。

汇和(C)消毒规则。 起初，似乎只有少数缺陷可以用这种方式来描述，但这个漏洞类很好地适合许多常见的安全缺陷，包括不同类型的缓冲区溢出和其他内存损坏缺陷、SQL和命令注入以及缺少授权检查。

污点式漏洞的一个突出例子是2014年[1]发现的开放SSL中的“心痛”错误。 图1显示了有问题的代码：整数有效载荷由从网络流（第11行）读取十六位整数的宏N2s定义)。 这个整数然后到达调用memcpy的第三个参数，而不需要进行任何类型的验证（第25行）。 特别是，不能保证有效载荷较小或等于源缓冲区pl的大小，因此，未初始化的堆存储器可以复制到缓冲区bp，然后通过在第29行的dtls_write_bytes发送到网络。

这个例子强调了识别污染类型漏洞的重要性，以及所涉及的一些困难。 首先，n2s是一个只在Open SSL的代码库中使用的宏，因此只有在考虑特定于项目的API函数时，才能在搜索模式中建模。 第二，第12行显示了为修补漏洞而引入的检查。 如果没有对代码库的深刻了解，有效载荷的消毒并不简单和难以分析。

为了足够详细地描述污染类型的漏洞，并在软件中搜索它们的化身，有必要检查信息如何从一个语句传播到另一个语句



无花果。 2：“心痛”bug的代码属性图的摘录。 数据流边、控制流边和语法边分别用红色、绿色和蓝色表示。

另一个，以及如何控制这种流动是由条件控制的。 因此，我们的方法基于代码属性图，这是一种源代码的中间表示，它将这些信息组合在一个结构中，并且可以使用图形遍历来挖掘漏洞。

B. 代码属性图

从软件工程和编译器设计的角度来看，程序代码存在着各种各样的表示。 例如，程序的结构可以描述为语法树，而程序语句的执行顺序则在控制流图中捕获。 这些标准表示中的几个可以表示为图形，因此Yamaguchi等人。 [63]建议使用图形数据库挖掘漏洞。 他们的方法的主要思想是构造一个所谓的代码属性图，这是一个属性图中程序结构、控制流和数据流的联合表示-许多图形数据库的本机存储格式。 这种联合表示使编程模式能够被编码为对图形数据库的查询，从而可以为危险编程模式的实例挖掘大量代码，从而缩小漏洞。

形式上，属性图是一个边缘标记的、归因于多图的[46]。 在实践中，这意味着键值对可以附加到节点和边缘以存储数据。 此外，边可以被标记以表示图中不同类型的关系。 例如，对于代码分析，可以为不同的语言元素创建节点，例如调用、谓词和参数。 然后，这些边可以通过标记边缘连接以表示执行顺序或数据流。

代码属性图利用这种通用的数据结构组合了三种现有的、众所周知的程序表示：抽象语法树，它表示程序结构是如何嵌套的，控制流图，它公开语句执行顺序，最后是程序依赖图，它使数据流和对依赖关系显式[见3，14]。在属性图中组合这些表示是可能的，因为所有这些表示都包含每个语句的指定节点，允许它们在这些节点上容易合并。

例如，图2显示了图1中函数的代码属性图的摘录。 图中包含每个程序语句的节点，包括输入和退出语句、对N2S的调用（第11行）、对memcpy的调用（第25行）和if语句（第12行）。 这些节点中的每一个都跨越一个由蓝色边表示的语法树，例如，将调用n2s分解为其语言元素。 此外，从程序依赖图获得的数据流边从第一个语句引入到第二个和第三个语句，以表明在第一个语句中产生的有效载荷的值达到第二个和第三个未修改的值，并在那里使用。 最后，函数中指示控制流的控制流边显示为绿色。 例如，一个无条件的控制流边将第一个语句连接到第二个语句，明确第二个语句在第一个语句之后执行。

一旦构建了不安全的编程模式，特别是油漆式漏洞的实例，可以在代码属性图中描述为遍历。 从一组种子节点开始，遍历根据节点的属性沿边移动的图。 遍历的输出是该移动终止的节点集。 例如，遍历可能会从对memcpy的所有调用开始，并沿着数据流边向后移动到宏N2s，从而提取类似于“心痛”漏洞的候选项（图1）。 此外，遍历可以利用控制流边来只选择memcpy和n2s之间的路径，其中没有对传播的变量有效载荷进行验证。

形式上，图遍历是映射一个集合的函数
节点到另一组节点。 因此，遍历可以自由地使用函数组合来产生新的遍历，从而可以用可重用的基本遍历来表示复杂的查询。 我们广泛地使用这种功能来构造对油漆式漏洞的搜索查询(参见第四节D)。 Yamaguchi等人详细介绍了漏洞发现的代码属性图和遍历。 [63].

III. 扩展代码属性图
程序间分析

代码属性图为基于模式的漏洞发现提供了丰富的信息，然而，它还没有考虑到过程间分析。 不幸的是，可用于推断搜索模式的信息是稀缺的

并且经常跨越几个函数，使得分析超越函数边界是可取的。 因此，我们寻求扩展代码属性图以获得类似于众所周知的系统依赖图的表示[23]但以适合使用图形数据库挖掘的格式。 我们可以通过扩展代码属性图来实现这一点，如下所示。 我们首先通过从参数到相应的callees的参数，以及从返回语句返回到调用站点，将调用站点与其callees之间的数据流明确化。 实际上，我们已经得到了一个表示函数之间调用关系的图，但是它编码的数据流信息是不精确的。 最重要的是，不考虑职能对其论点所作的修改，也不考虑这些修改的影响随着数据沿着呼叫链返回。

在下面，我们描述了一种通过使用后控制器树检测参数修改来改进这个初步图的方法(第三-A节)，这两种方法都是为了处理对源代码可用的函数的调用，以及对于只能观察到调用者的库函数(第三-B节)。 我们继续通过图传播这些信息，以获得用于推断搜索模式的代码属性图的最终程序版本(第三-C节)。

A. 添加后分母树

对于我们检测参数定义的启发式方法，确定语句是否总是在另一个语句之前执行的能力是至关重要的。 不幸的是，合并到属性图中的现有经典程序表示不允许很容易地确定这一点；控制流图只指示语句是否可以在另一个语句之后执行，程序依赖图的控制依赖关系仅限于暴露必须在执行语句之前进行评估的谓词。

支配树和后支配树[见3，8]，两个经典的程序表示可以从控制流图导出，非常适合解决这个问题。 对于控制流图和程序依赖图，这些树包含每个语句的节点。 这些节点通过边缘连接以表示优势，这一概念与强制性语句执行顺序的分析密切相关。

在控制流图中，节点d控制另一个节点n，如果n的每条路径都必须首先通过d。通过将每个节点链接到其直接控制器，我们得到了一个控制器树。 类似地，节点p后支配另一个节点n，如果来自n的每条路径都必须通过p。通过再次将每个节点的即时后支配者连接在一起，我们得到了一个后支配者树。

作为示例，图4显示了图3中正在运行的示例的函数条的后控制器树。与所有后控制器树一样，树植根于退出节点，因为CFG中的所有路径最终都会通过退出节点。 然而，由于边缘只存在于节点到它们的直接后控制器，只有谓词y<10和调用foo(x，y，z)连接到退出节点。 相反，

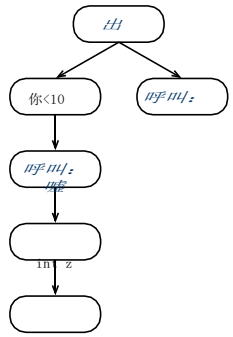
```
int bar(int x, int y){
      int z;
      嘘(&z);
      如果(y<10)
        FOO(x, y, &z);
 }

 int boo(int*z){
      *z=得到();
 }

在MOO(){
      =得到();
      英特b=1;
      酒吧(a, b);
 }

 吴(){
      在=1;
      =得到();
      酒吧(a, b);
 }
```

无花果。 3：调用sinkfoo的运行示例



无花果。 4：函数栏的后定位器树

对boo的调用立即由谓词支配，而它立即支配语句intz。

这两种数据结构都很方便，如果我们需要快速确定代码库中的语句是否总是在另一个语句之前或之后。 此外，由于每个语句都存在指定的节点，通过在语句节点之间添加适当标记的支配边缘，可以很容易地将这些结构与现有的代码属性图合并。

B. 检测参数修改

一旦后控制器树可用，我们就可以使用它们来检测导致修改其参数的函数调用-这是在编译器设计[3]中表示为定义参数的过程。 虽然对于常见的库函数，如从POSIX标准读取或recv，这个问题可以通过提供注释来解决，但内部API，如“心痛”漏洞中存在的N2S宏，不被识别为输入源。

因此，一般来说，我们需要假设对库函数的调用是否导致修改其参数是未知的，因此，我们可以充当数据源。实际上，对于库函数的所有直接和间接调用者，参数

定义可能无法正确检测。 例如，考虑POSIX库函数的读和写，它们都将指向内存缓冲区的指针作为它们的第二个参数。 仅从函数签名，就无法确定read修改缓冲区的内容而write不修改缓冲区的内容。 然而，这是一个重要的差异，直接影响代码属性图中的数据流边。

为了解决这个问题，我们继续如下。 对于没有源代码的每个遇到的函数，我们通过基于以下两个检查计算一个简单的统计量来确定它是否可能定义它的参数。

1) 我们检查本地变量声明是否通过数据流到达参数，而不需要进行容易识别的初始化，例如赋值或对构造函数的调用。
2) 我们检查后控制器树中从函数到本地变量声明的路径不包含另一个语句，该语句也通过数据流直接连接到变量声明。

计算满足这两个条件的调用站点的分数，我们假设一个参数是由对函数的调用定义的，如果这个分数超过了一个定义的阈值。 对于我们的实验，我们将这个阈值固定在10%，以避免错过任何攻击者控制的来源。 借助这种简单的启发式方法，我们重新计算了没有源代码的函数的所有调用者的代码属性图中的数据流边。 图3说明了我们的启发式：本地变量z在第2行中声明，没有明显的初始化。然后把它作为第3行和第5行的一个参数传递给boo和foo，分别。 而对于函数boo，假设是合理的
它初始化z，这对于函数foo来说是不正确的
在boo之后调用，可能已经初始化z。

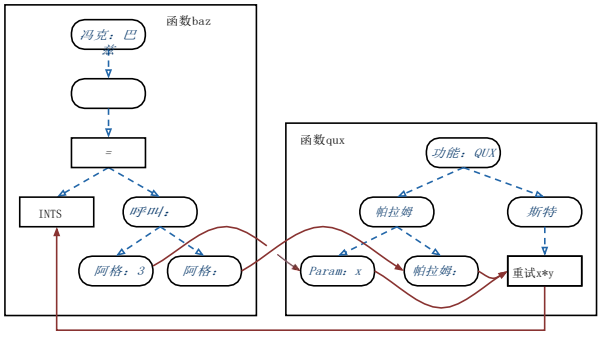
C. 数据流信息的传播

除了通过库函数检测参数定义的问题外，参数定义可能间接发生，即函数调用的任何函数都可能负责参数定义。因此，在函数中识别数据流的源而不下降到其所有Callees是无效的。 例如，考虑图3所示的代码片段。 函数foo的参数z首先在第2行中定义，然后在调用的函数boo中的第9行中重新定义。 因此，有一个数据流从源获取到函数foo。

考虑到间接参数定义，我们可以
沿着调用链传播数据流信息。 为此，我们通过分析每个函数的主体并检查其任何参数是否存在，确定每个函数（有可用的源代码）是否进行了参数定义
(a) 在函数内部定义，(B)此定义通过控制流到达退出语句。 然而，只有当函数的数据流边已经考虑到由它调用的任何函数执行的参数定义时，这种方法才有效。 因此，在分析函数本身之前，我们分析函数的所有Callees，并将上一节中提出的启发式方法应用于库函数。

算法1数据流 重新计算

```
1：程序FIXDATA FLOWEDGES(V)
2:    对于v∈v做
3:       f_v ← 假的                    标记节点不固定
4:    为v ∈有
5:       FIX NODE(v)
6：程序FIX NODE(V)7:
      如果F_v=是真的8:
         返回false
9:    f_v←真，你←假
10:   对于c∈CALLEES(V)做                把所有的卡莱都修好
11:      你←你∨FIX NODE(C)
12:   如果你=真的                       dv需要更新
13:      更新ATAFLOW(V)
14:      返回真实
15:   返回 假的
```



(a) 过程间代码属性图。

```
1  无效Baz（无效）{                    整数(intx，inty){返回x* y；  4
2      =QU X（3，7）；                 }                            5
3  }                                                                6
```

(b) 来电者和来电者的代码片段。

无花果。 5：函数baz和qux的过程间代码属性图。 语法边缘显示为虚线，数据流边显示为实线。

算法1通过递归地更新代码属性图的节点来实现这一思想。 特别是，节点v的数据流边被固定使用过程FIXNODE，其中属性f_v确保没有节点被访问两次。 该算法使用预序遍历下到图中，即在处理当前函数（第13行）之前，所有Callees都被更新（第11行）。 完成后 算法1，可观测参数定义和 由此产生的间接数据流在图表中进行了说明。

作为生成的过程间代码属性图的示例，请考虑代码片段和图5中给出的图形。 图是由抽象语法树的节点构造的，其中它的边缘要么反映语法（虚线），要么反映数据流(实线）。 控制流边在本例中没有显示。

从概念上讲，这种代码的过程表示直接来源于Horwitz等人引入的经典系统依赖图(SDG。 [23]，却调到了

使用图形数据库查询进行处理，并使用语法和优势信息进行增强。 这个结构非常适合于建模和搜索漏洞，正如我们在下面的章节中所说明的那样。

IV. 漏洞搜索模式的推断

配备了一个用于过程间分析的代码属性图，我们现在已经准备好解决为污点式漏洞提取搜索模式的问题。 从安全敏感的sink开始，例如内存函数memcpy，我们的目标是以图形遍历的形式生成搜索模式，从而能够发现数据流到sink中的漏洞。 为了有用，这些查询需要足够通用，以编码代码的模式，而不是特定的调用。 此外，它们需要精确地捕捉跨函数的数据流，以便能够正确地跟踪单个参数的定义。 最后，生成的查询应该易于被从业者理解和修改，允许纳入额外的领域知识。

为了生成具有这些特性的搜索模式，我们实现了以下四步过程，它结合了静态代码分析、机器学习和签名生成的技术（图6）。

1) *定义图的生成。* 对于所选接收器的每个调用，我们通过分析代码属性图生成定义图。 类似于程序间程序切片[23]，这些图形紧凑地编码参数定义和消毒，尽管在一个两级结构中，专门创建，以方便地枚举可行的调用(第四-A节)。
2) *解压和聚类。* 然后，我们将定义图解压缩为单个调用，并对所有调用站点进行聚类，以获得公共参数定义的模式(第四-B节)。
3) *创造卫生覆盖。* 接下来，我们通过从数据流中添加潜在的消毒来扩展生成的模式，即流中限制参数值的所有条件(第四-C节)。
4) *生成图形遍历。* 最后，我们使用分析平台Joern(第四-D节)以适合于高效处理的图形遍历的形式表示推断的搜索模式)。

在下面的章节中，我们将更详细地描述这些步骤中的每一个，并举例说明它们。

A. 定义图的生成

虽然源代码包含关于如何调用函数的有价值的信息，特别是关于如何定义和清理它们的参数的信息，但这些信息通常分布在几个不同的函数中，并隐藏在不相关的代码中。 为了有效地利用这些信息，需要一个源汇表示，它只编码参数的定义和消毒，同时丢弃所有其他语句。 为了解决这个问题，我们生成了一个图表示对于每个源汇系统，可以很容易地从我们的代码属性图中计算。 这种表示，在整个论文中被称为定义图，编码所有观察到的参数定义组合及其相应的消毒。 因此，定义图是从相应的过程程序片的节点的一个精心选择的子集创建的[见23，60]，其中只包含与确定油漆样式漏洞的搜索模式相关的节点。 定义图允许轻松地枚举可行的参数初始化，就像通过解决相应的上下文无关语言可达性问题来完成过程间程序切片一样[见44]。

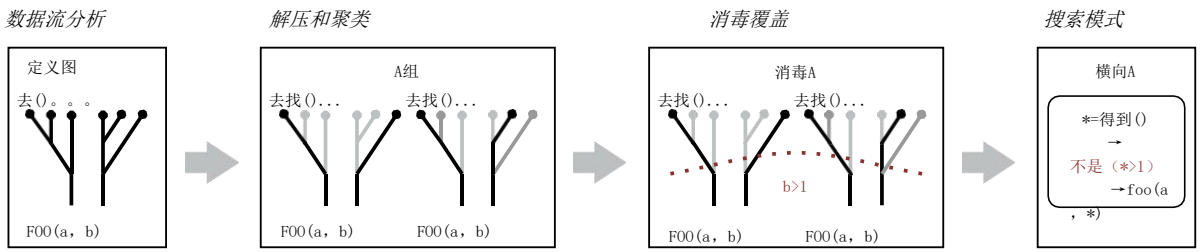
我们首先对单个函数进行局部建模，然后将各自的图组合起来，对函数交互进行建模。

1) 局部函数建模：在函数的边界内，使用程序切片技术可以很容易地确定影响调用的语句和所涉及的变量。这允许我们创建一个层次表示，它捕获涉及调用中使用的变量的所有定义语句以及控制调用站点执行的所有条件。为了说明这种表示的构造，我们将图3中对函数foo（第5行）的调用视为选定的接收器。 从这个调用站点开始，我们使用以下规则通过传递代码属性图来构造表示：

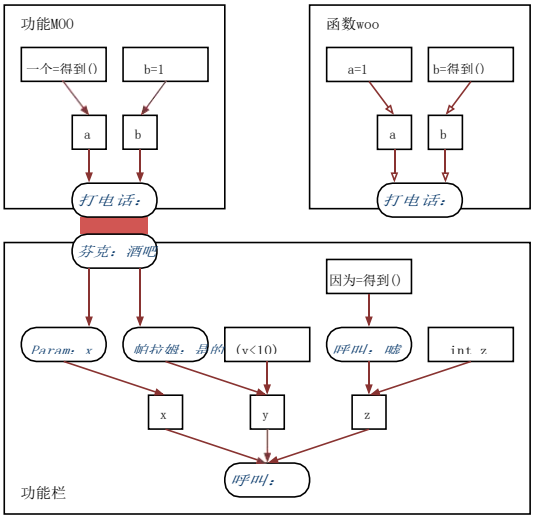
- 对于选定的接收器，我们首先遵循输出语法边缘到其参数。 在示例中，我们扩展foo以达到参数x、y和z。
- 对于这些参数和定义它们的所有语句，我们然后遵循连接的数据流和对照依赖边缘来揭示定义语句以及控制调用站点foo的条件。 例如，定义intz和条件y<10就是这样发现的。
- 最后，我们使用过程边从定义我们的任何变量的所有调用移动到相应的函数体，在那里我们确定了影响所选接收器参数的进一步定义语句。 在这个例子中，对boo的调用是由这个规则发现的，引导我们得到定义语句*z=得到()。

对于以这种方式到达的每个参数，我们将所有各自的调用站点视为接收器，并递归地应用这些规则为通过数据流连接到初始接收器的每个函数获得一棵树。 这些树可以很容易地从代码属性图中构造，使用深度优先遍历，该遍历使用实现给出的三个规则的展开函数。

2) 定义图：有了函数的树表示，我们已经可以分析参数定义及其在函数中的消毒，但是调用者仍然没有被探索。 在示例代码中，这意味着我们揭示了z的局部变量定义，而参数x和y不能跟踪到函数边界。 不幸的是，在局部构造过程中，下面的参数到参数边缘的简单而直观的解决方案



无花果。 6：概述我们推断漏洞搜索模式的方法。 该方法从选定的接收器(Foo)开始，自动构造捕获数据流中的源(GET（）)和消毒(B>1)的模式。



. 无花果。 7：图3中运行示例的函数foo的定义图，其参数由MoO定义，图的第二个实例化用虚线表示。

树产生不可行的定义组合，如Reps[44]详细讨论的那样。例如，在我们的例子中，这个简单的解决方案将生成inta的组合{=得到()，intb=得到()。 但是，这个组合是无效的，因为第一个定义只发生在moo调用bar时，而第二个定义发生在woo调用bar时。 因此，这些定义永远不会合并出现。

这是一个经典的程序间程序分析问题，例如，可以通过制定相应的上下文无关语言可达性问题[44]来解决。 另一个解决方案是简单地确保函数的参数节点在遍历图时总是一起展开。 例如，当扩展x的参数节点时，y的节点也需要扩展。 此外，需要确保两个节点都用来自同一个调用站点的参数展开，在我们的示例中，要么是woo，要么是moo。

作为解决方案，我们只是通过建模整个函数的相互作用而不是参数来将参数联系在一起。 的定义图实现了这一思想。 与局部树建模函数相比，定义图的节点不仅仅是过程代码属性图节点的子集，而是表示整个树。 因此，定义图是两个层次的结构，它结合了用于局部建模函数以表示它们的调用关系的树。 例如，图7显示了示例代码中调用foo的定义图。 形式上，我们可以定义这些定义图如下。

定义1调用站点c的定义图G=(V，E)是一个图，其中V由为c和所有直接和间接调用者的那些树在本地建模的树组成。对于每个a，b，如果由b表示的调用表示的函数，则E中存在从a到b的边。 ∈

B. 减压和聚类

我们现在有了一个源汇表示，它使参数的定义及其消毒变得明确。 因此，我们试图确定定义图中反映参数定义的常见组合的模式。 给定一组任意的定义图，例如函数memcpy的所有调用站点的所有定义图，我们使用机器学习技术生成类似定义组合的集群及其消毒器，设计成易于转换为图形数据库遍历(见第四-D节)。 我们在以下三个步骤中构造这些集群。

1) 定义图的分解：虽然定义图只表示单个接收器，但它可能编码参数定义的多个组合。 例如，那个图7中的定义图包含组合{intz、=获取()、b=1}以及组合{intz、a=1、b以压缩形式获取。 幸运的是，枚举存储在定义图中的所有组合可以使用一个简单的递归过程来实现，如算法2所示，其中$[v_0]$表示a名单含有只是的节点v_0运算符+表示列表连接。

定义图的节点是树，其中每个参数定义的组合对应于这些节点中表示调用链的子集。 因此，该算法从根节点r(V)开始，简单地将当前树与所有可能的调用链结合起来，即所有树列表

算法2解压缩定义 图表

```
1：程序(G)
2:    返回RDECOMPRESS(G, r(V))
3：程序规则(G：=(V, E), v0)
4:    =Øp
5:    d←    不是    t REES(v0)
6:    如果D=那么Ø
7:       返回{[v0]}
8:    对于d∈d做
9:       对于L∈RDECOM PRESS(G, d)做
10:         R←R∪0]+1)
11：返回R
```

在导致此树的代码库中遇到。 由于这一步，我们得到了由S表示的所有观察到的参数定义组合的集合。

2) 聚类Callees和类型：Callees和具有类似名称的类型通常实现类似的功能。 为了考试-

请注意，函数malloc和realloc都是相关的
使用分配，而strcpy和strcat处理复制
定义的组合，即使没有一个参数是
使用完全相同的Callee或类型定义。 为了实现这个目

我们确定了相似的Callees和类型的簇
构建漏洞的搜索模式。

特别是，我们独立地对每个参数的Callees和类型进行聚类。 由于我们对紧凑表示感兴趣，我们应用了完全链接聚类，这是一种以创建紧凑的对象组和易于校准而闻名的技术[见4]。 链接聚类需要在考虑的对象上定义一个距离度量，我们使用Jaro距离来完成这个任务，因为它是专门为比较短字符串[25]而设计的。 Jaro距离将两个字符串的相似性量化为0到1之间的值，其中1表示精确匹配，0表示绝对不同。 分析人员可以通过指定Jaro距离的最小相似性来控制集群内相似字符串的需要程度。 我们发现，对于0之间的聚类参数，聚类是相对稳定的。9和0。并将参数固定为0。为我们所有的实验。 由于这一步，我们得到了每个参数的一组CALLE和类型簇。

3) 对定义组合的聚类：对参数定义的解压组合进行聚类比对Callees和类型进行聚类涉及得更多，因为这些组合是复杂的对象，而不是简单的字符串。 我们的目标是比较定义组合在它们附加到参数上的定义，尽管在某种程度上对Callees和类型的名称的细微差异具有鲁棒性。 我们通过使用一个广义的词袋模型来实现这一点，并将组合映射到由上一步计算的聚类跨越的向量空间[见45]。

在下面，让我们假设一个具有单个参数的接收器，并让S表示组合集，而C是CALLE和类型簇的集合。 然后，对于每个组合s∈S，
我们可以确定簇$C_s \subseteq C$认为它的定义

都包含在里面。 然后我们代表每个 在一个空间中，每个维度与CSWE的一个簇相关联，通过定义一个映射，可以实现这一点.

ϕ：$S \rightarrow \{0, 1\}^n$其中c‘th坐标Φ_c是由

$$\Phi_c = \begin{cases} \text{如果} c \in C_s \\ \text{否则为} 0 \end{cases}$$

而n是CALLE和类型簇C的总数。

对于具有多个参数的接收器，我们独立地对每个参数执行此操作，并简单地连接结果向量。 作为一个例子，让我们考虑一个定义组合s，其中第一个参数是通过对malloc的调用初始化的，而第二个参数被定义为size_t类型。 则对应的向量有如下形式。

{查尔[52]，乌查尔[32]，。。。}

阿格。 1

使用这种向量表示，我们现在可以使用
再次链接聚类，以获得参数定义的类似组合集群。 作为聚类的距离函数，我们选择城市块距离，因为它为测量向量中是否存在簇提供了一种直观的方法。 我们在整个实验中对聚类参数使用固定值为3，这意味着集群内的调用可能在多达三个条目中有所不同。 由于这一步骤，我们获得了类似组合的集群，即构成代码库中存在的模式的类似调用组。 这些集群的大小可以用来对这些模式进行排序，以便优先检查主导模式：较大的集群表示由许多单独调用支持的强模式，而小集群表示不那么清晰的模式，仅由很少的调用支持。

C. 创建排序规则重叠

在上一步中生成的集群已经可以用来生成搜索模式，指示用于接收器的主要参数组合。 然而，它们不编码消毒模式。 为了实现这一点，我们继续为表示每个参数的典型消毒的参数定义模型创建覆盖。 为此，我们利用包含在任何定义图中的所有条件中的信息。 正如callee和类型匹配的情况一样，我们寻求对条件的制定方式的轻微变化保持健壮。 为此，我们利用每个条件在代码属性图中表示为语法树的事实。 类似于分类和类型的分组方式，我们将这些树映射到向量，并使用链接聚类对它们进行聚类。 特别是，我们使用基于邻域哈希核[16,22]的显式树嵌入。

设表示条件集合，每一种条件都用语法树表示。 然后，我们将每个条件t映射到向量如下：首先，语法树中的内部节点被其类型属性的哈希值标记，例如乘法、一元表达式或赋值，而叶节点则被其代码属性的哈希值标记，通常是标识符、运算符或文字。 第二，正在传播的符号被重命名为var。 这样，条件就不取决于标识符的名称。数字、关系和等式运算符也使用正则表达式进行规范化。最后，通过计算其邻域哈希作为其子节点标签AS的函数来考虑每个节点v的邻域

$$h(v)=r(l(v)\oplus \sum_{z\in C_v} l(z))$$

其中l(V)是节点v的标签，r()表示单比特旋转，表示节点标签上的逐位异或操作，最后是C_v 是v的子节点。实际上，我们为每个语法树获得了一组哈希值，我们可以用来表示它。 我们定义了映射Φ：0，1" 从条件到n维向量，其中n是 不→{ } 不同的哈希值和条件c∈T

$$\Phi_j(c)=\begin{cases}\text{如果c含有h(V)=j的节点v}\\ \text{否则为0}\end{cases}.$$

在对defnition图中的每个条件执行此映射时，我们再次使用链接聚类，使用固定参数为2的城市块距离，。 我们存储上一节中计算的每个组合集群中使用的所有条件的集群标识符，最后将它们附加到搜索模式中。

D. 图形遍历的生成

生成的集群增强了卫生覆盖完全表达搜索模式，现在可以映射到图形遍历，以挖掘漏洞。 为了实现这一点，我们为基于模式的分析平台Joern构建了一个通用模板，该模板可以捕获缺少的参数消毒，并且可以很容易地实例化，以表示不同的搜索模式，以油漆样式的缺陷。

1) 遍历模板：图8显示了查询语言Gremlin中的模板遍历。 为了实例化模板，我们需要定义敏感接收器的名称、每个参数的数据源描述以及它们的消毒描述。 这些被称为argiSource和argiSanitizer，其中我表示参数编号。 遍历通过确定接收器的所有调用站点来进行，然后使用被污染的Args遍历并按顺序未检查每个接收器。.

遍历受污染的Args用于查找接收器是否符合源描述(argi Source)。 它通过首先生成IV-A节中描述的相应定义图来实现这一点。 不需要减压

```
获取呼叫（接收器）

得到了阿尔格斯(
        [arg1Source, ..., argn Source]
)

未经检查(
        [arg1Sanitizer, .argnSanitizer]
)
```

无花果。 作为查询语言Gremlin中的图形遍历的油漆样式漏洞的8：模板。

然后，它通过检查每个描述是否存在至少一个匹配语句来确定调用站点是否可以完成参数描述。

这个步骤大大减少了需要进一步分析的调用站点的数量，但是，我们还不能确定调用站点是否与参数描述相匹配。 为了实现这一点，根据算法2解压缩特定的定义图。有了定义组合，检查参数描述是否匹配就变得微不足道了。 最后，我们返回所有与描述相匹配的定义组合，并将它们传递给未检查的遍历，以进行消毒分析。

遍历未检查确定所有调用站点，其中至少有一个参数没有根据该调用站点进行消毒

消毒说明。 该函数通过根据各自的消毒剂描述检查定义图中的每个条件来进行。

2) 模板实例化：我们通过将定义和条件分别转换为参数描述和消毒描述来实例化来自集群的查询。 回顾每一种论点都有一套数据来源集群和条件，这些只是必须以一种适合安全分析员容易理解的形式加以总结。 为此，我们通过确定签名生成中通常执行的最长公共子序列，从这些集群中生成正则表达式[见40]。 然后，可以使用生成的图形遍历来挖掘错误代码，并允许分析人员进行细化。

V. 评价

我们将分两步对五个流行的开源应用程序的源代码进行评估。 首先，我们执行一个受控的实验，在这个实验中，我们评估了我们的方法为已知的漏洞生成遍历的能力，并测量了这些如何减少要检查的调用站点的数量。其次，我们评估了我们的方法的能力，以帮助一个真实的代码审计流行的媒体播放器VLC，在那里我们发现了几个以前未知的漏洞。 我们将我们的方法实现为代码分析平台Joern版本0.3.1的插件，使用库快速集群[37]进行聚类。 为了让其他研究人员复制我们的结果，我们将我们的插件作为开源提供[2].

[2]https://github.com/fabsx00/querygen

项目	版本	构成部分	一行代码	脆弱性	敏感的水槽	#呼叫站点
Linux	3.11.4	司机密码	6,723,955	CVE-2013-4513	copy_from_user	1715
打开SSL	1.1.0f	整个图书馆	378,691	CVE-2014-0160	memcpy	738
洋泾浜	2.10.7	整个项目	363,746	CVE-2013-6482	阿托伊	255
VLC	2.0.1	整个项目	555,773	CVE-2012-3377	memcpy	879
多普勒(Xpdf)	0.24.1	整个图书馆	227,837	CVE-2013-4473	sprintf	22

表一：具有已知污染类型漏洞的五个开源项目的数据集。 该表还列出了每个漏洞的敏感汇和我们的方法推断的遍历次数。

	正确的来源	正确的消毒方法	#列车	一代时代	执行时间	减少[%]
CVE-2013-4513	c	c	37	142.10秒	10.25秒	96.50
CVE-2014-0160	c	c	38	110.42秒	8.24秒	99.19
CVE-2013-6482	c	c	3	20.76秒	3.80秒	92.16
CVE-2012-3377	c	c	60	229.66秒	20.42秒	91.13
CVE-2013-4473	c		1	12.32秒	2.55秒	95.46
平均						94.90

表二：为发现五种颜色样式的漏洞而减少对审计的代码。 对于最后一个漏洞，由于呼叫站点数量少，没有正确的消毒剂被引用。

A. 受控实验

为了评估我们的方法在受控设置中生成程序代码中真实漏洞查询的能力，我们分析了五个流行的开源项目的安全历史：Linux内核、密码库OpenSSL、即时信使Pidgin、媒体播放器VLC以及文档查看器Evince和Xpdf使用的呈现库Poppler。 对于这些项目中的每一个，我们确定了一个最近的污染类型的漏洞和相关的敏感接收器。 表一提供了此数据集的概述，显示了项目及其版本、易受攻击组件及其包含的代码行。 此外，还显示了由其CVE标识符表示的漏洞、相关的敏感接收器和接收器的调用站点数。 我们现在详细地描述这些污点式的漏洞。

- *CVE-2013-4513(Linux)*。 攻击者控制的名为size_t类型计数的变量作为第三个参数传递给接收器copy_from_user而不被消毒，从而触发缓冲区溢出。
- *CVE-2014-0160(开放SSL“心痛”)*。 源n2s定义的无符号int类型的变量有效载荷作为第三个参数传递给memcpy，而不被检查，导致缓冲区被过度读取。
- *CVE-2013-6482（洋泾浜）*。 未读的字符串从攻击者控制的源xmlnode_get_data读取并传递到sinkatoi，而不需要进行消毒，从而可能导致NULL指针被取消引用。
- *CVE-2012-3377(VLC)*。 数据缓冲区p_stream->p_headers的长度取决于攻击者通过函数realloc控制的分配，并且在不验证可用缓冲区大小的情况下到达对memcpy的调用，从而导致缓冲区溢出。
- *CVE-2013-4473(Poppler)*。 攻击者控制的字符串dest File Name被复制到char类型的本地堆栈缓冲区路径Name中[1024]使用函数sprintf而不检查其长度，从而导致基于堆栈的缓冲区溢出。

我们继续为所有这些汇生成遍历。 表二总结了我们的结果，显示了为每个漏洞生成的遍历次数，以及我们的方法是否能够生成一个既表示正确的源又表示消毒剂的遍历。 它还显示了从代码生成遍历所需的时间，以及遍历的执行时间(以秒为单位。 最后，显示了在使用生成的遍历作为漏洞的健壮签名时不必检查的调用站点的百分比（减少百分比）。

我们的方法在所有情况下为各自的参数源生成正确的描述，并且在除一个情况外的所有情况下都正确地消毒。在CVE-2013-4473的情况下，没有返回消毒说明。 在这种情况下，只有22个调用站点可用，使得使用统计方法很难推断消毒剂描述。 尽管如此，我们的查询大大减少了要检查以定位漏洞的调用站点的数量，平均可以跳过94.9%的调用站点。

最后，表三显示了源和汇的推断正则表达式。 在这些正则表达式中，来自漏洞描述的攻击者控制源的名称是清晰可见的。 此外，除了bug描述中的消毒模式外，在某些情况下还可以识别额外的消毒剂。 例如，该方法确定来自源n2s的memcpy的第一个参数通常与NULL进行比较，以确保它不是NULL指针。 对于执行多个消毒剂的参数，只显示一个。

	CVE-2013-4513	CVE-2014-0160	CVE-2013-6482	CVE-2012-3377	CVE-2013-4473
下沉	copy_from_user	memcpy	阿托伊	memcpy	sprintf
论点1	.*	.*	.*xmlnode_get_。*	.*配置。*	.*查尔[。* \].*
论点2	.*康斯特。*恰恰。*r*.*	.*	.*	.*	.*
论点3	.*size_t。*	.*n2s。*	.*	.*	.*
消毒液1个	–	.*赛姆（==|！=）没空。*	.*赛姆。*	.*赛姆。*	–
消毒剂2	–	–	–	–	–
消毒剂3	.*赛姆。*(d+)。*	.*赛姆。*\+(\d+)。*	–	–	–

表三：包含在五种颜色样式漏洞的搜索模式中的正则表达式，其中sym在运行时被跟踪符号替换。 对于最后一个漏洞，没有推断出消毒剂。

B. 案例研究：心脏出血的脆弱性

在这个案例研究中，我们展示了我们的方法是如何成功地为第二节中提出的“心痛”漏洞生成搜索模式的，作为一个典型的漏洞的例子。 我们使用我们的方法在Open SSL版本1.1.0f中为安全敏感的sink memcpy生成模式，这是库的最后一个版本，容易受到这个特定错误的影响。在strcpy、strcat和sprintf等函数中，memcpy是与缓冲区溢出漏洞最常见的函数之一[见5，11]。

我们首先使用第三-B节中提出的启发式方法来发现定义它们的参数的库函数。 图9a显示了发现的库函数名和参数编号。 我们手动验证其中的每一个，以发现除了一个都是正确推断的。 对于错误识别的memset的第三个参数，我们发现它通常是形式size of（缓冲区），其中缓冲区是一个在没有事先定义的情况下到达memset的变量。稍微调整我们的启发式来解释size of的语义（通过抑制它的参数）也解决了这个问题，只给我们留下了正确推断的数据源。

功能	定义
fgets	1. 争论
sprintf	1. 争论
memset	1. 争论
写作	3. 争论
memcpy	1. 争论
memset′	3.争论
n2s	2. 争论
n2l	2. 争论
c2l	2. 争论

(a) 图书馆的功能

有规律的表达
*n2s。 *
*记忆体。 *
*斯特伦。 *
。 *int.*len。 *
。 *in arg。 *
。 *大小。*
*未署名。 *
。*int。 *
*长。 *

(b) 推断的数据来源

无花果。 9：(a)绘制由我们的启发式识别的库函数；(b) 为memcpy的第三个参数的数据源推断正则表达式。

然后从代码中推断查询导致生成38个查询，其中14个查询为memcpy的第三个参数指定了一个源。 这些都是特别有趣的，因为第三个参数指定要复制的数据量，因此攻击者控制的情况是主要的候选
缓冲区溢出。 图9b包含第三个参数的9个源，显示攻击者控制的源n2s
尤其是。 由于n2s是攻击者控制的唯一来源，因此只需要执行图10中所示的单个遍历。

```
arg3Source=源匹配(’。*n2s。*’);

arg2洗手液={it，符号->条件匹配(“。*不==|！=。*“，符号)
};

arg3Sanitizer={it，符号->条件匹配(“。*%s。*\+(\d+)。*“，符号)
};

获取呼叫(“memcpy”)
包含Args([任何，任何，arg3Source])
未检查([ANY_OR_NONE，arg2Sanitizer，arg3Sanitizer])
```

无花果。 10：生成的遍历编码易受伤害的编程模式，从而导致心痛的漏洞。

查询编码从攻击者控制的数据源n2s到memcpy的第三个参数的信息流。 此外，它强制执行两个卫生规则。 首先，需要检查传递给memcpy的第二个参数是否为NULL指针，其次需要在包含整数的表达式中检查第三个参数。显然，分析人员可以很容易地修改这些规则以提高精度；然而，为了本案例研究的目的，我们采用了原样的遍历。即使没有细化，遍历只返回738个调用站点（0.81%），如表四所示。 其中，两个完全对应于“心痛”的脆弱性。

C. 案例研究：VLC媒体播放器的脆弱性

在这个案例研究中，我们说明了我们的方法是如何发挥作用的
在识别五个以前未知的关键作用

文件名	功能
ssl/d1both	处理心跳
SSL/s3clnt.c	获得密钥交换
SSL/s3clnt.c	获得新的会话票证
SSL/s3srvr.c	获取客户端密钥交换
SSL/t1lib.c	SSL解析clienthellotlsext
SSL/t1lib.c	处理心跳
密码/缓冲区	BUF_memdup

表四：生成的查询返回的七次点击。 脆弱的功能是阴影。

穿越	文件名	功能	线	CVE标识符
横向1	模块/services_discovery/sap.c	解析SDP	1187	CVE-2014-9630
横向1	模块/stream_out/rtpfmt.c	RTP安抚Xiph配置_	544	CVE-2014-9630
横向1	模块/访问/ftp.c	ftp发送命令	122	CVE-2015-1203
横向2	模块/codec/dirac.c	密码	926	CVE-2014-9629
横向2	模块/codec/schroedinger.c	密码	1554	CVE-2014-9629

表五：由遍历提取的调用站点。 所有这些呼叫站点都很脆弱。

```
=源匹配('。*查尔[。*伦\+。* \]。 *')arg3Src={源匹配('。*size_t。*')
（它）||
                源匹配('。*斯特。*伦。*')（它）}

获取呼叫("memcpy")
包含Args([arg1Src, ANY_SOURCE, arg3Src])
```

无花果。 11：Traversal为memcpy的第一个参数识别堆栈内存的动态分配。

漏洞在VLC，一个流行的开源媒体播放器。 为此，我们选择了为sink memcpy生成的两个遍历，它们看起来特别有趣，因为它们直接编码危险的编程实践。 第一个查询（如图11所示）描述了对memcpy的调用，其中第一个参数被定义为char类型的本地堆栈缓冲区。 此外，在定义内动态计算大小。 这本身已经构成了一个有问题的编程实践，因为无法验证可用的堆栈内存是否允许执行此分配。特别是，如果要分配的内存量由攻击者控制，并且随后使用memcpy将内存复制到缓冲区中，攻击者可能会损坏内存并利用内存执行任意代码。

运行此查询返回三个调用站点，所有这些站点都有问题。 特别是，图13显示了脆弱函数rtp_packetize_xiph_config其中，在第14行中，变量len被计算为攻击者控制字符串的长度。 然后用于在第15行分配堆栈缓冲区b64，最后在第16行将len字节复制到缓冲区。 通过在64位Linux平台上触发无效的内存访问，成功地确认了此漏洞的存在。

图12显示了第二个有趣的查询：在这种情况下，memcpy的第二个参数来自与正则表达式匹配的源。*去吧。*。 这对应于VLC媒体播放器中直接从媒体读取的宏族
可能由攻击者控制的文件。 要复制到缓冲区的数据量直接取决于的情况

```
arg20Source=源匹配('。*去吧。*'); arg21Source=source
Matches('。*uint。*_t。*');

获取呼叫("memcpy")
[ANY_SOURCE, ANY_SOURCE,
                {arg20Source(it)&arg21Source(it)}]
```

无花果。 12： 遍历以识别memcpy的第三个参数定义为。*去吧。*.

```
1  国际rtp_packetize_xiph_config(sout_stream_id_t*id,
2                                  康斯特查尔*FMTP, int64_t
3                                  i_pts)
4  {
5    如果(fmtp==空)返回
6      VLC_EGENERIC;
7
8    /* 从fmtp中提取base64配置*/char*启动=strstr(fmtp,
9    "配置="); 断言(启动！=NULL);
10   开始+=size of（"配置="）-1; char*结束=strchr
11   （开始，';'）; 断言(结束！=无）;
12   size_t len=端启动;
13   charb64[len+1]; memcpy(b64,
14   start, len); b64[len]= '\0' ;
15   // [...]
16 }
17
18
19
```

无花果。 13：以前使用第一次遍历发现的未知漏洞。

攻击者控制的整数是缓冲区溢出的常用源，因此我们选择查询。

表五显示了查询返回的两个函数，这两个函数都是易受攻击的。 特别是，如图14所示，源文件模块/codec/dirac.c中的函数Encode会导致缓冲区溢出：32位变量len由攻击者控制的源GetDWBE在第5行初始化，并用于第7行的分配。 不幸的是，在分配之前直接将固定值size of(eos)添加到len中，导致整数溢出。 实际上，为缓冲区p_extra分配的内存太少。 最后，在第10行中，len字节被复制到大小不足的缓冲区中，导致溢出。

总之，我们确定了5个以前未知的漏洞，可以利用这些漏洞执行任意代码。

```
静态block_t*编码(encoder_t*p_enc, picture_t*p_pic)
{
 如果(！p_enc->fmt_out。p_extra){
   // [...]
   uint32_t=获得DWBE(p_block->p_buffer+5);
   // [...]
   p_enc->fmt_out。p_extra=malloc(Len+size of(EOS)); 如果(！p_enc-
   >fmt_out。p_extra)
     返回NULL;
   memcpy(p_enc->fmt_out.p_extra, p_block->fmt_outp_buffer, len);
   // [...]
 }
```

无花果。 14：以前使用第二次遍历发现的未知漏洞。

此外，我们通过选择两个有前途的自动生成查询来实现这一点，说明了我们的方法作为安全分析人员审查漏洞代码的工具的实际优点。

VI. 限制

通过自动推断的搜索模式发现先前未知的漏洞证明了我们方法的优点。 然而，我们在下面讨论的一些局限性。

首先，由于我们的推理设置，如果代码库的主要部分缺乏适当的消毒，我们的方法无法从数据流中识别相应的消毒规则，从而无法生成准确的搜索模式。 幸运的是，这一限制不适用于成熟的软件项目，这些项目在处理用户控制的数据时广泛使用了消毒处理。

其次，我们的方法假设数据只通过函数参数和返回值从调用者传递给Callee。 共享资源，如全局变量或共享内存，不采用我们的方法建模。 因此，我们无法描述攻击者在这些资源之间将数据传播到接收器的污染式漏洞。修改程序代码属性图以解释这种类型的数据流似乎是涉及的，但可能。 我们将这一修改作为今后工作的延伸。

第三，到目前为止所讨论的工作只显示了我们的方法在识别C代码典型的漏洞方面的适用性，例如无效的内存访问。 虽然原则上，我们的方法应该适用于其他几种漏洞类型，特别是典型的Web应用程序缺陷，但这一点仍有待证明。 特别是，使我们的方法适应不同的语言需要仔细处理特定于语言的属性。

最后，我们的方法没有完全恢复软件的控制流程。 特别是，动态调用目前没有解决。 同样，我们的方法不能描述基于并发执行函数的漏洞，例如许多在无使用后的安全缺陷。 这种限制在处理代码属性图和动态分析技术（如动态油漆跟踪或符号执行）之间的耦合并不简单，也可能有益于此。

VII. 相关工作

在软件中发现漏洞的方法的开发是安全研究中的一个长期课题，它涉及到广泛的方法和技术。 在讨论相关工作时，我们将重点放在一些方法上，这些方法也旨在协助安全专家对软件进行审计，而不是取代她。

a) 基于查询语言和注释的方法：与我们的工作密切相关的方法是使分析员能够使用查询语言或注释搜索漏洞。 例如，Martin等人的方法。
[35]和Lam等人。 [30]都使用描述性查询语言来建模代码和查找软件缺陷。

同样，Vanegue等人。 [56]实验与扩展静态检查[15]作为HAVOC工具的一部分，并测试其性能在一个大的代码审计。 此外，还提出了几种基于安全类型系统的发现信息流漏洞的方法[见21、47、50]。 特别是，Jif编译器[38,39]执行类型检查，以允许对带有注释的Java版本执行安全策略。 此外，Jif还实现了一种类型推断算法，以减少所需的用户定义注释的数量。

最后，Evans和Larochelle[13]使用C的注释作为查找代码中脆弱模式的手段。 虽然我们的方法有着相似的动机，但它的不同之处在于它自动填充搜索模式，因此分析师只需要定义一组安全敏感的汇来开始审核未知的代码库。

b) 推断编程模式和规范：手工分析代码是一项繁琐而耗时的任务。 作为一种补救措施，已经提出了几种方法，利用统计方法、机器学习和数据挖掘技术来加速这一过程。 为此，sev-eral方法从API的代码、修订历史[33]和先决条件[例如，7、41、55]中自动推断编程模式[例如，19、32、58]和安全规范[例如，28、34、54]。 一项相关的研究遵循了一种更有原则的方法，通过建模和推断安全策略[例如，6、36、52、57]来发现信息流漏洞。 与我们的方法相似，许多这些方法都基于语法树和代码片以及结合语法、控制流和数据依赖关系的表示[例如，27，29]。

Engler等人。 [12]首先指出，源代码中的缺陷通常可以与违反隐式引入的系统特定编程模式相联系。 它们提供了一种方法，可以自动将用户提供的规则模板定制到特定的系统，并演示其识别系统代码缺陷的能力。 与这项工作密切相关，Kremenek等人。 [28]进一步表明，基于因子图的方法允许自动组合不同的证据来源，以生成违规检测器的规范。

与漏洞发现更密切相关，Livshits等人。 [34]介绍了Merlin，一种基于因子图的方法，它为Microsoft从Web应用程序中输入信息流规范。 NET框架。 梅林的一个重要限制是，它只模拟函数之间的信息流，因此，源、消毒剂和汇总是被假定为函数的调用。 虽然对于典型的Web应用程序漏洞，这种假设在许多情况下是成立的，但无法以这种方式检测到缓冲区溢出或空指针检查等漏洞的缺界检查。 相反，我们的方法非常适合对这些检查进行编码，因为消毒剂来自任意语句，允许对声明和条件中的模式进行建模（参见第五节）。 同样，山口等人。 [62]目前的查奇，一种方法，以检测缺失的检查，也能够处理消毒的任意条件-

问题。 不幸的是，该方法对从业者是不透明的，因此不可能控制或改进检测过程。 与Merlin和Chucky相比，源、消毒剂和汇被表示为正则表达式，作为遍历的一部分，使分析人员很容易调整它们以进一步改进规范。 最后，一些作者使用相似性度量来确定类似于已知漏洞[17,24,42,61]的漏洞。

c) 基于动态分析的方法：大量的研究集中在探索漏洞发现的动态代码分析上。 最显著的是黑盒引信[例如，43、53]和白盒引信技术[例如，18、20、59]。 这些方法与我们的工作是正交的，因为它们在运行时探索源汇系统中的数据流。 虽然不是专门为帮助人类分析人员而设计的，但白盒模糊可能会补充我们的方法，并有助于探索攻击者可以访问代码的哪些部分，以进一步缩小漏洞。

VIII. 结论

发现软件中未知的漏洞是一个具有挑战性的问题，通常需要大量的人工审计和分析工作。 虽然我们的方法通常不能消除这一努力，但搜索模式的自动推断显着地加速了对大型代码库的分析。 借助这些模式，从业者可以将分析集中到相关的代码区域，并更容易地识别污染类型的漏洞。 我们的评估表明，在“心脏出血”漏洞的情况下，要审计的代码量平均减少了94.9%，甚至进一步减少了94.9%，这表明自动生成的搜索模式可以精确地建模油漆风格的漏洞。

我们的工作还表明，精确方法的相互作用，如静态程序分析，与相当模糊的方法，如机器学习技术，为漏洞发现提供了富有成果的基础。 虽然精确的方法对软件的特性提供了丰富的观点，但人类分析师很难理解这种观点的复杂性。 模糊方法可以帮助过滤这个视图-在我们的设置中通过搜索模式-从而指导从业者在审计代码漏洞

报告脆弱性

我们已经与供应商合作，以修复所有的漏洞，作为我们的研究的一部分。 即将出现的版本不应再包含这些缺陷。

感谢

我们确认DFG在DEVIL项目(RI2469/1-1)下提供的资金)。 我们还要感谢谷歌，特别是我们的赞助商蒂姆·科诺，通过谷歌学院研究奖支持我们的工作。 最后，我们感谢我们的牧羊人安德烈·萨贝尔菲尔德和匿名评论员的宝贵反馈。

参考资料

[1] 心脏出血的虫，2014年。http://heartbleed.com/，

[2] Shellshock漏洞。 http://shellshockvuln.com/，2014.

[3] A.阿霍，R.塞蒂和J.乌尔曼。 *编译器原理、技术和工具。 爱迪生-韦斯利，1985年。*

[4] M.安德伯格。 *应用聚类分析学术出版社，纽约，美国，1973年。.*

[5] c.Anley，J.Heasman，F。林德纳和G。里查特。 *壳牌的手册：发现和利用安全漏洞。 约翰·威利和儿子，2011年。*

[6] Backes M.，B.Kopf和A.Rybalchenko。 信息泄露的自动发现和量化。 *在检察院。 IEEE安全与隐私研讨会，2009年。*

[7] r.。 张，波德古斯基和J。杨。 利用依赖图挖掘软件中被忽略的条件。 *IEEE软件工程交易，34（5）：579-596，2008。*

[8] K.D.Cooper，T。j.哈维和K。肯尼迪。 一种简单，快速的优势算法。 *软件实践与经验，4：1-10，2001年。*

[9] 科瓦M.，V。Felmetsger，G.Banks和G.Vigna。 静态检测x86可执行文件中的漏洞。 在检察院。 *计算机安全应用年度会议(ACSAC)，2006年。*

[10] J.Dahse和T。霍尔兹。 仿真内置PHP特性进行精确静态代码分析。 在检察院。 网络和分布式系统安全研讨会(NDSS)，2014年。

[11] m. 多德，J。 麦克唐纳和J。 舒。 *软件安全评估的艺术：识别和防止软件漏洞。 培生教育，2006年。*

[12] 恩格勒D.y.陈先生，夏先生，周先生，周先生。 异常行为：推断系统代码错误的一般方法。 在检察院。 2001年ACM操作系统原则专题讨论会。

[13] D.埃文斯和D.拉罗切尔。 使用可扩展的轻量级静态分析来提高安全性。 *IEEE软件，19（1）：42-51，2002。*

[14] J.Ferrante，K.J.Otstein和D。沃伦。 程序依赖图及其在优化中的应用。 *关于编程语言和系统的ACM交易，9：319-349，1987年。*

[15] c. 弗拉纳根，K。 r。 m。 利诺，M。Lillibridge，G。 纳尔逊，
j. B.撒克逊和罗斯塔塔。 对java进行扩展静态检查。 在ACM计划通知，第37卷，第234-245页，2002年。

[16] 加斯康H.，F。山口，D.Arp和K.Rieck。 使用嵌入式调用图对Android恶意软件进行结构检测。 在检察院。 关于人工智能和安全的ACM讲习班，2013年。

[17] f.Gauthier，T。拉沃伊和梅洛。 揭示访问控制的弱点和缺陷与安全不协调的软件克隆。 在检察院。 计算机安全应用年度会议(ACSAC)，2013年。

[18] p.甲状腺，M。y.列文和D。 莫尔纳。 圣人：

用于安全测试的白盒引信。 ACM通信，55（3）：40-44，2012年。

[19] N.格鲁斯卡，A.Wasylkowski和A.Zeller。 学习6000个项目：轻量级跨项目异常检测。 在检察院。软件测试和分析国际研讨会(ISSTA)，2010年。

[20] I.Haller，A.Slowinska，M.Neugschwandtner，和H.博斯。 显示溢出：引导引信查找缓冲区边界违规。 在检察院。 2013年USENIX安全研讨会。

[21] 海因策N.和里克。 SLAM演算：具有保密性和完整性的编程。 在检察院。 *1998年关于方案拟订语文原则的ACM专题讨论会。*

[22] S.希多和卡希马。 线性时间图核。 *在检察院。 IEEE国际数据挖掘会议(ICDM)，2009年。*

[23] S.霍维茨，T。众议员和丁·宾克利。 使用依赖图进行过程间切片。 在检察院。 ACM国际编程语言设计和实现会议(PLDI)，第35-46页，1988年。

[24] J.张，阿格拉瓦尔和德布鲁姆利。 Re De Bug：在整个OS发行版中找到未匹配的代码克隆。 在检察院。 *IEEE安全与隐私研讨会，2012年。*

[25] M.雅罗。 1985年佛罗里达州坦帕人口普查所采用的记录联系方法的进展。 *美国统计协会杂志，84（406）：414-420，1989年。*

[26] 约万诺维奇N.，克鲁格尔和基尔达。 Pixy：用于检测Web应用程序漏洞的静态分析工具。 在检察院。IEEE安全与隐私研讨会，2006年。

[27] D.A.Kinloch和M.Munro。 理解C程序使用组合C图表示。 在检察院。 *国际软件维护会议(ICSM)，1994年。*

[28] t.克雷梅内克，P。二希，G.Back，A.Ng和D.Engler。 从不确定性到信念：推断内部的规范。在检察院。 2006年操作系统设计与实施专题讨论会。

[29] J.克林克和格斯内廷。 验证测量软件作为切片和约束求解的应用。 *信息和软件技术，40（11）：661-675，1998年。*

[30] M.林书豪，J。鲸，五。利弗希茨B.，医学博士。 马丁，
D.Avots，M.Carbin和C.Unkel。 上下文敏感的程序分析作为数据库查询。 在检察院。 数据库系统原则专题讨论会，2005年。

[31] S.列基，B。 股票和约翰先生。 2500万流以后：基于DOM的XSS的大规模检测。 在检察院。 *ACM计算机和通信安全会议，2013年。*

[32] 李Z.和Y。周。 PR-Miner：自动提取隐式编程规则并检测大型软件代码中的违规行为。 在检察院。欧洲软件工程会议(ESEC)，第306-315页，2005年。

[33] 利夫希特B.和T。齐默曼。 Dyna Mine：通过挖掘软件修订历史来找到常见的错误模式。 在检察院。欧洲软件工程会议(ESEC)，第296-305页，2005年。

[34] 利夫希茨B.，A。v.Nori，S.K.Rajamani和A.Banerjee。 梅林：显式信息流问题的规范推理。在检察院。 ACM国际编程语言设计与实现会议，2009年。

[35] 马丁M.，利夫希茨和林M.。 使用PQL：Program Query Language查找应用程序错误和安全缺陷。 在检察院。 2005年ACM面向对象编程、系统、语言和应用会议。

[36] I.Mastroeni和A.Banerjee。 使用抽象域完整性建模解密策略。 *计算机科学中的数学结构，21（06）：1253-1299，2011。*

[37] D.穆纳。 快速集群：R和Python的快速分层、聚集集群例程。 *统计软件杂志，53（9）：118，2013年。*

[38] A.。迈尔斯。 Jflow：实用的多静态信息流控制。在检察院。 1999年方案拟订语文原则问题交流小组讨论会。

[39] a. c。 迈尔斯，L。 郑，S。 兹丹斯维奇，S。 冲，还有
N.尼斯特罗姆。 JIF：Java信息流。 *软件发布。 http://www. 位于cs。 康奈尔。 Edu/jif，2001年。*

[40] J.Newsome，B.Karp和D.Song。 多形图：自动生成多态蠕虫的签名。 *在检察院。 IEEE安全与隐私研讨会，2005年。*

[41] H.A.Nguyen，R.Dyer，T。阮N.和拉詹。 挖掘大规模代码语料库中API的前提条件。 在检察院。 2014年ACM软件工程基础国际研讨会(FSE。

[42] j.佩维，F。舒斯特，C。罗索，伯恩哈德，还有
t.霍尔兹。 利用语义签名进行二进制程序中的bug搜索。 在检察院。 计算机安全应用年度会议(ACSAC)，2014年。

[43] A.Rebert，S.K.Cha，T。Avgerinos，J.Foote，D。沃伦，
G.Grieco和D.Brumley。 优化引信种子选择。 在检察院。 2014年USENIX安全研讨会。

[44] t.代表。 通过图形可达性进行程序分析。 *信息和软件技术，1998年。*

[45] K.Rieck，C.Wressnegger和A.Bikadorov。 莎莉：在向量空间中嵌入字符串的工具。 *机器学习研究杂志(JMLR)，13（11月）：3247-3251，11月。* 2012.

[46] M.罗德里格斯和P。纽鲍尔。 图遍历模式。 *图数据管理：技术和应用，2011年。*

[47] 萨贝尔菲尔德A.和迈尔斯。 基于语言的信息流安全。 *IEEE通信选择领域杂志，21（1）：5-19，2003。*

[48] p.Saxena，S.Hanna，P。波桑坎和宋医生。 亚麻：

在丰富的Web应用程序中系统地发现客户端验证漏洞。 在检察院。 网络和分布式系统安全研讨会(NDSS)，2010年。

[49] E.施瓦茨，T。Avgerinos和D.Brumley。 所有你想知道的动态污点分析和向前符号执行（但可能是害怕问）。 在检察院。 IEEE安全与隐私研讨会，2010年。

[50] U.尚卡尔，K。塔尔瓦，J.S.福斯特和D。瓦格纳。 使用类型限定符检测格式字符串漏洞。 在检察院。 *2001年USENIX安全研讨会。*

[51] S.儿子，K S.麦克金利和V。什马蒂科夫。 角色转换：当你不知道检查是什么时，找到丢失的安全检查。 在检察院。 面向对象编程系统语言和应用国际会议，2011年。

[52] v.斯利瓦斯塔瓦，邦德，麦金利，还有 v.什马蒂科夫。 安全策略Oracle：使用多个API实现检测安全漏洞。 在检察院。 *ACM国际编程语言设计和实现会议，2011年。*

[53] M.萨顿，格林和P。阿米尼。 引信：钝力脆弱性发现。 艾迪森-韦斯利专业，2007年。

[54] l.谭某，张某，马某，W。熊和Y。周。 自动ISES：自动推断安全规范和检测违规行为。 在检察院。 2008年USENIX安全研讨会。

[55] S.和T谢。 阿拉丁：挖掘替代模式来检测被忽略的条件。 在检察院。 *自动化软件国际会议的报告*工程(ASE)，第283-294页，2009年。

[56] j.凡盖，布隆伯格和拉希里。 使用扩展静态检查器进行实际的无功安全审计。 在检察院。 IEEE安全与隐私研讨会，2013年。

[57] j. a.沃恩和S.Chong。 表达性解密策略的引用。在检察院。 IEEE安全与隐私研讨会，2011年。

[58] H.Vijayakumar 、 X.Ge 、 M.Payer 和 T 。 杰 格 。 JIGSAW：通过推断程序员的期望来保护资源访问。在检察院。 2014年USENIX安全研讨会。

[59] t.王，T。魏，林，和W。Zou。 国际范围：使用符号执行自动检测x86二进制文件中的整数溢出漏洞。在检察院。 *网络和分布式系统安全研讨会(NDSS)，2009年。*

[60] m.吝啬鬼。 *程序切片。 在检察院。 国际软件工程会议，1981年。*

[61] f.山口，M.Lottmann和K.Rieck。 使用抽象语法树进行广义漏洞外推。 *在检察院。 计算机安全应用年度会议(ACSAC)，2012年。*

[62] f. 山口，C。 女同性恋。 加斯康和K。 里克。Chucky：暴露源代码中缺少的漏洞发现检查。 在检察院。 ACM计算机和通信安全会议，2013年。

[63] f.Yamaguchi，N.Golde，D.Arp和K.Rieck。 用代码属性图建模和发现漏洞。 在检察院。 IEEE安全与隐私研讨会，2014年。