第5章

移动终端漏洞静态分析技术

漏洞静态分析技术是指不需要运行代码而直接对代码进行漏洞挖掘的方法。因此，该方法适用于完整的或不完整的源代码、二进制代码及中间代码片段。事实上，漏洞静态分析的主要方法均来自软件分析领域，其关键技术和核心算法与软件分析方法中的技术和算法相同，二者的主要区别是目标不同。软件分析的目标是发现软件缺陷，保障软件质量；漏洞静态分析的目的是发现漏洞（漏洞属于软件安全缺陷，是软件缺陷的一种），保障软件安全性。下面介绍在漏洞挖掘领域中经常使用的软件静态分析方法的原理及过程。

5.1 流分析

人们很早就开始对代码进行分析，并且提出了多种分析方法（如控制流分析和数据流分析等），统称流分析。需要指出的是，流分析得出的结果都较为粗略，适用于编译优化，但对于安全缺陷的检查还有所欠缺，需要更精确的代码语义信息。

5.1.1 流分析的分类及定义

根据代码分析的目的，可将流分析分为两大类：一类是控制流分析，另一类是数据流分析。控制流分析是要得出代码中控制流走向的信息，即控制流图。控制流图是对代码执行时可能经过的所有路径的图形化表示，通过对代码中的分支、循环等关系的分析来获得代码的结构关系。数据流分析是要得出程序中数据流动的信息，即程序中变量的相关信息，例如，可到达的变量定义、可用的表达式、别名信息、变量的使用及取值情况等。总的来说，控制流分析关心的是代码的控制结构信息，数据流分析关心的是代码中的数据信息。

根据代码分析的范围，可将流分析分为过程内的流分析和过程间的流分析两大类。过程内的流分析着眼于一个单独的函数（过程），每次分析一个函数，不考虑函数之间的影

响。过程间的流分析包括多个函数，分析时考虑函数间的关系。过程内的流分析技术相对较为成熟，而过程间的流分析尚有许多未解决的问题。

下面给出流分析的一些基本定义。

对于程序语句的序列，可以把它们划分成最小的单位是基本块，基本块是满足以下两个条件的最长的语句序列。

(1) 控制流只能从基本块的第一条语句进入。

(2) 控制流只能从基本块的最后一条语句出去。

当程序语句被划分成基本块后，可以用一张图来表示基本块之间的控制关系，称为控制流图(Control Flow Graph，CFG)。CFG 的节点是基本块。CFG 的边由以下规则添加：从基本块 B 到基本块 C 有一条边，当且仅当程序的控制流可以从 B 出来而后进入 C。一个程序点定义为 CFG 上的一点：每两条语句之间有一个程序点，每条语句前后各有一个程序点。程序中所有变量的值称为程序的状态。

通常的数据流分析不要求已知每个变量的具体值，而是把状态映射到一个抽象的域里，称为数据流值域。例如，人们关心变量在程序点处是否有定义，变量的值就被映射为 true 或 false，状态就变成了一个位向量，每位对应于一个变量：如果该位是 1 表示该变量有定义，如果该位是 0 表示该变量没有定义。

通常，状态被映射到的数据流值域是一个格(Lattice)，数据流分析操作的就是格里的元素。格是一种代数结构，包括一个集合 L，以及 meet 操作和 join 操作，满足以下 4 个性质。

(1) 封闭性：对于 L 中所有的 x 和 y，都存在着唯一的 z 和 w 属于 L，使得 x meet $y=z$ 和 x join $y=w$ 成立。

(2) 交换性：对于 L 中所有的 x 和 y，x meet $y=y$ meet x 以及 x join $y=y$ join x。

(3) 结合性：对于 L 中所有的 x 和 y，(x meet y)meet $z=x$ meet(y meet z)以及(x join y) join $z=x$ join (y join z)。

(4) L 中存在两个唯一的元素：底记为 bottom，顶记为 top。使对于 L 中所有的 x，x meet bottom=bottom 以及 x join top=top。

在数据流分析中用到的大多数格都以位向量作为它们的元素，meet 和 join 分别为按位与和按位或。

每条语句都会对状态产生一定的影响，把在它之前的状态映射到它之后的新状态。于是每条语句就对应着一个传输函数(Transfer Function)，这个函数将一个数据流值映射到另一个数据流值。因此传输函数是一个从格到它本身的函数，$f: L \to L$。它刻画了程序语句对状态的影响。对于同一条语句，在不同的数据流分析问题中，对应于不同的传输函数。

一个函数在格 L 中的不动点是一个元素 z，z 属于 L，使 $f(x)=z$。

5.1.2 数据流分析的分类和求解方法

数据流分析的问题可以从 3 方面进行分类。

(1) 数据流分析所提供的信息。

(2) 所使用的格，格中元素的含义，以及定义在格上的函数。

(3) 信息流的方向。前向问题,信息流的方向与程序执行的方向相同;后向问题,信息流的方向与程序执行的方向相反。

下面描述编译优化用到的最重要的几类数据流分析问题,需要指出的是,进行数据流分析的对象程序的存在形式有很多种,比较常见的程序的表示形式有接近于源代码层次的抽象语法树(Abstract Syntax Tree,AST)和基于静态单赋值(Static Single Assignment,SSA)的接近于汇编的三地址码中间形式。在不同的表示形式中进行数据流分析的方法和难度不同,所以在实际应用中应该根据需要选择不同的程序表示形式分析不同的数据流问题。

下面列出一些常见的数据流分析问题。

(1) 到达定义。到达定义确定可达变量使用地点对变量的赋值,这是一个前向问题。使用位向量的格,每位对应于一个变量定义。

(2) 可用表达式。可用表达式确定在某个程序点处可用的表达式,一个表达式在一个程序点处可用意味着在表达式被求值的地点和该程序点之间的所有路径上没有对表达式中的变量赋值。这是一个前向问题,位向量中的每位对应一个表达式的定义。

(3) 活变量。活变量确定在某个程序点处某个变量是否会在该程序点到程序之间被用到,这是一个反向问题,可以让每个变量使用对应一个位,也可以让每个变量对应一个位。

(4) 前向暴露的使用。前向暴露的使用确定哪些变量使用能够被一个特殊的变量赋值到达,这个问题和到达定义是对偶问题。它也是前向问题,每位对应一个变量的使用。

(5) 复制传播分析。复制传播分析确定在一个复制赋值 x=y 和一个对 x 的使用之间是否有对 y 的赋值。这是一个前向问题,每位对应一个复制赋值。如果没有对 y 的赋值,就可以把对 x 的使用直接替换成 x。

(6) 常数传播分析。常数传播分析确定在一个常数赋值 x=c 和一个对 x 的使用之间是否有其他对 x 的赋值,这是一个前向问题,每位向量对应一个常数赋值。

(7) 部分冗余性分析。部分冗余性分析确定被重复执行了两次的计算。这是个双向问题。每位对应一个表达式计算。

上述 7 种数据流分析问题在编译优化里是最重要也是使用最多的几种。

解决数据流分析的方法有很多种,包括强连通区域法、迭代法、路径压缩法、图语法、消去法、语法导向法、结构分析法、区间分析法等。上述方法中最常用的方法是迭代法,迭代法的主要步骤:初始化数据流信息状态及工作列表,然后依次处理工作列表中的节点,更新各节点信息,同时把被它影响的节点加入工作列表中,由于传输函数在数据流信息的格中有不动点,所以迭代法是可以终止的。

5.1.3　过程内的流分析

在编译器发展的早期,由于编译优化的需要,人们开始对被编译的程序进行分析,试图得到关于程序的数据流信息。例如,对下面这个代码片段:

```
x=3;
*p=4;
```

```
y=x;
```

如果要对 x 做常数传播(即在使用 x 的地方用常数值 3 替代),需要知道 p 是否有可能指向 x,只有在 p 不会指向 x 的情况下,才能把程序变换成

```
x=3;
*p=4;
y=3;
```

以分析的精确度来分类,过程内的数据流分析可以分为流不敏感分析、流敏感分析、路径敏感分析 3 种。

(1) 流不敏感分析一般给出的是一个函数整体的数据流信息,如一个函数有可能修改哪些变量。

(2) 流敏感分析给出一个函数的 CFG 上每点对应的信息。

(3) 路径敏感分析对函数 CFG 上每点可能会给出多个信息,因为沿着不同的路径到达同一个程序点很可能会产生不同的状态信息。路径敏感分析保留这些不同的信息,而流敏感分析在控制流汇聚的程序点处会将不同分支传进来的状态汇聚成一个状态。

在编译优化领域,人们主要关心流不敏感分析和流敏感分析。一方面因为在编译时进行的分析需要很快完成,而路径敏感分析由于代价比较高,基本不被使用;另一方面,由于编译优化需要的信息是保守的,即这些信息在任何情况下都要成立,而路径敏感分析的信息不具有这一特点,所以不被使用。然而以安全缺陷为目标的静态分析需要的是精确信息,路径敏感分析方法就非常适合。

5.1.4 过程间分析

前面介绍的过程内的流分析关心的是单个函数如何进行分析得到代码的信息。本节对过程间的流分析做初步介绍。过程间的流分析面对的问题的规模和难度都比过程内的流分析大得多,因此过程间的流分析中的很多问题还没有完善的算法。

(1) 控制流分析。过程间的控制流分析关注的主要问题是函数调用图的构建。程序的函数调用图定义:以程序中的每个函数为顶点,若函数 A 调用了函数 B,则存在一条以 A 为起点 B 为终点的边,这些顶点和边就组成了函数调用图。

对于一些语言,函数调用图较容易构建,只要扫描程序,碰到函数调用语句进行相应的记录。但是对于 C 和 C++ 语言,由于函数指针和虚函数的存在,函数调用图就较难构建。一般情况下只能构建一个近似的函数调用图。

(2) 数据流分析。过程间的数据流分析主要解决函数调用语句的副作用问题。

在进行过程内的流分析时,需要模拟每条语句的作用。对于一般语句,如赋值、分支等,它们的作用是明确的。但是对于函数调用语句,它们的作用是未知的,不能从函数调用语句本身获得。如果不对函数本身进行分析,就只能对函数调用语句的作用做最保守的假设,或完全忽略它。这种处理方法的分析过程必然很不精确。为了弥补这个缺陷,有下面两种处理方法。

最简单的方法就是每次碰到函数调用就直接进入被调用的函数进行分析,分析完后,

再回到原来的函数中继续往下分析，这样做有一个严重的问题就是代价太高。如果每次碰到一个函数调用都进入被调函数，函数调用深度会越嵌越深，分析的复杂度会呈指数上升。所以这种就地展开函数的方法在一般情况下不可行。

另一种可行的方法是对每个函数只分析一次，分析完后记录它的总结信息。下次碰到调用该函数的语句，直接读取总结信息，作为该函数调用语句的作用指导分析。目前这是一种通行的做法。

根据函数的总结信息和函数调用地点的关系，可以把过程间的流分析分为上下文不敏感分析和上下文敏感分析。在上下文不敏感分析中，一个函数的总结信息是不变的。无论该函数在哪里以什么参数被调用，都只能得到关于该函数的同样的总结信息。在上下文敏感分析中，根据调用地点和调用参数可以得到一个函数的不同的总结信息。很容易看出，上下文敏感分析要更精确，但同时它的代价也更高。

尽管有了函数总结信息这样一个一般的思路，但如何设计函数的总结信息仍然是个没有完全解决的问题。在传统的程序分析中，函数的总结信息都非常简单，例如，函数的总结信息是该函数可能修改的变量的集合，或函数的总结信息是该函数是否返回一个常数值。

而对于软件正确性及安全性分析，这样简单的函数的总结信息用处不大，必须设计算法得到更精确、更复杂的函数的总结信息。从单个函数到过程间的流分析碰到的根本性困难是函数调用语句如何处理，本质上是如何让信息在函数之间进行传播。基本的解决思路是在函数调用处使用函数的总结信息。过程间的流分析一直都是程序分析中最困难的问题，至今没有完全解决，所以这里只给出一些基本的介绍。

5.2　符号执行方法

5.1 节介绍了以格为模型的流敏感数据流分析，本节详细介绍在软件漏洞检测中非常重要的以程序执行状态为模型的路径敏感分析方法，也就是符号执行。

传统的数据流分析主要从程序中提取布尔信息，信息表示只有 true、false 两种，例如，变量是否被某个函数修改了，一个表达式的值在某个程序点是否可用，表达式之间是否具有别名关系等。获取这样的布尔信息只需要对程序建立一个比较粗略的模型。例如，可以建立一个存储程序表达式布尔信息的表格，然后把程序的操作语义进行简化，变成对布尔信息的操作。由于传统的数据流分析所服务的目标是编译优化，这样的布尔信息也就够用了。但是对于以漏洞挖掘为目标的代码分析，仅仅知道布尔信息是不够的，必须对程序运行时的状态建立新的模型，获取更丰富的信息，才能够进行错误检查或服务于其他的目的。

因此，需要以程序运行的完整状态为模型进行分析，包括变量的值及其他相关信息。在函数开始时假设所有的输入变量是未知的，给它一个符号值。这个符号值具有和变量相同的类型。随着程序操作的进行，这些符号值会在数学运算和赋值运算的作用下衍生出新的符号值，也会通过程序路径中的条件表达式得到关于这些符号值的约束。有了每个程序点状态的符号描述和关于符号的约束，把要检查的错误也表示成一个关于变量值的条件表达式，再通过检查这个条件表达式在当前约束下是否可满足，就可以判定是否存

在安全缺陷。

符号执行的目标是把程序转化成一组约束，同时检查程序模拟执行过程中的状态是否出错。这组约束中既包含程序中的路径条件，也包含要求程序满足的正确性条件或程序员给出的断言。符号执行方法也是在程序的 CFG 上使用 WorkList 算法进行遍历，但由于操作模型是完整的程序状态，分析的过程就复杂很多。下面对比一下流敏感数据流及符号执行之间在状态数量上的区别。

从 5.1 节可知，流敏感数据流分析给每个程序点关联上一个状态（格中的一个元素）。如果是前向分析，一条语句后面的状态由语句前面的状态经过语句所对应的传输函数确定。在 CFG 上分支合并的地方，为了保证一个程序点只有一个状态与之相关联，不同分支传过来的状态要根据数据流分析的类型应用格中的 meet 或 join 操作合并成一个状态。在流敏感数据流分析中，分析完产生的包含状态的 CFG 和原先的 CFG 结构是一样的，只是在每个程序点处关联上了一个状态，这个状态是程序沿着所有可能的执行路径到达该点的状态的总体近似。

在面向代码正确性或安全性的分析中，如果也使用流敏感分析，在控制流汇聚的节点合并前面的状态，则会造成信息的过分丢失，从而无法得到想要的结果。因此，在分析过程中，控制流汇聚的节点不合并状态，保留所有状态，即对每个 CFG 的节点关联上不止一个状态。不同路径传播的状态在控制流汇聚的地方不会合并，而是保持原样继续传播。这样每个节点的前驱和后继形成了一条单独的程序执行路径。符号执行就是这样一种路径敏感分析。

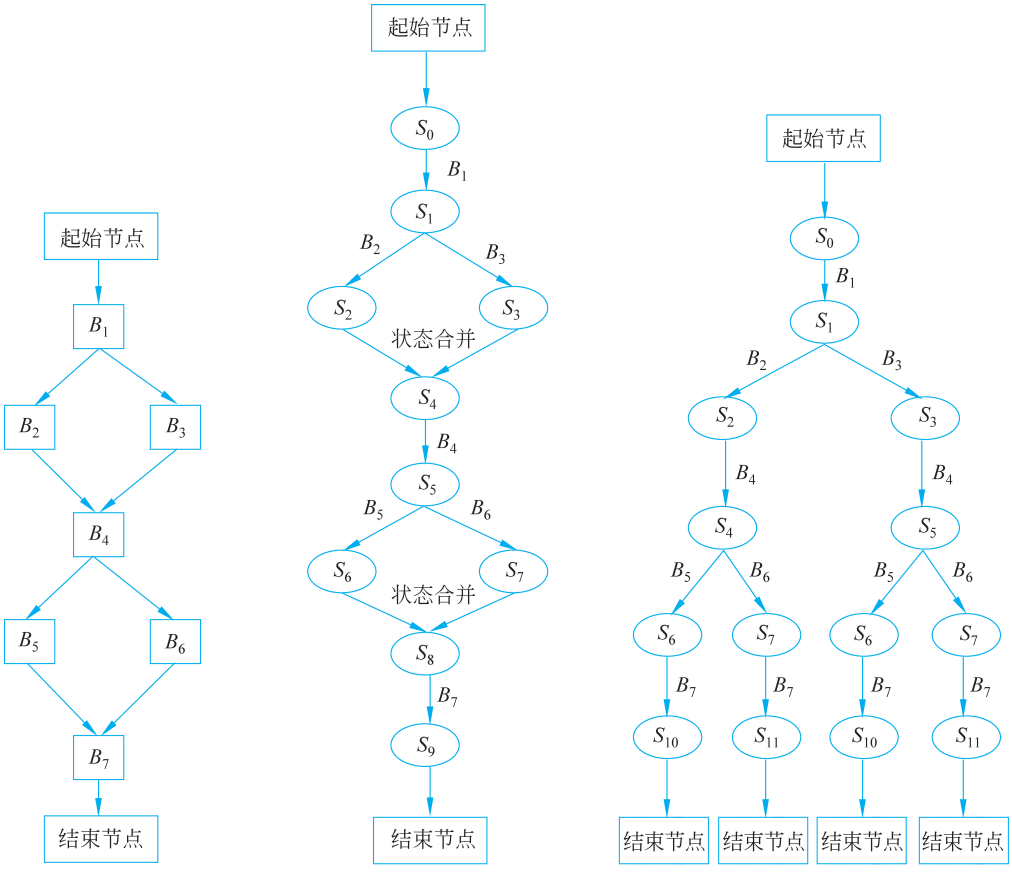
路径敏感分析中每个程序点处的状态是程序沿着一条真实的执行路径到达该处的状态，如果把不同的（程序点，状态）对看成不同的节点，即程序点相同、状态不同的节点也看成不同的节点，那么分析完之后产生的 CFG 节点比原先的 CFG 节点要多，把这个新图称为扩展图（Exploded Graph）。

原始 CFG 定义为（N，E，entry，exit）。其中，N 为所有顶点的集合；E 为所有边的集合；entry 为唯一起始节点；exit 为唯一结束节点。

带状态的 CFG 定义为（N′，E′，init，end）。其中，N′为所有状态的集合；E′为所有迁移的集合，即 E′＝{（a，b），如果 a 的程序点到 b 的程序点有一条边在 E′中}；init 为初始状态；end 为结束状态。

扩展的 CFG 定义为（N″，E″，（entry，init_state），EOP）。其中，N″为生成的（程序点，状态）的集合；E″＝{（a，b），如果 a 的程序点到 b 的程序点有一条边在 E″中}；（entry，init_state）为起始节点和初始状态，EOP 是（exit，end）的集合。因为不同的路径到达 exit 的状态不同，所以 EOP 中也包含了多个节点。扩展图的节点是（程序点，状态）的集合，边则是根据原始 CFG 的控制结构得到的边，起始节点还是一个，结束节点则变成了多个。

原始 CFG、带状态的 CFG 以及扩展的 CFG 之间的对比如图 5-1 所示。其中，图 5-1（b）由流敏感分析生成。椭圆代表程序点处的状态，边代表基本块关联的传输函数。图 5-1（c）由路径敏感分析生成。椭圆代表（程序点，状态）对，注意状态在控制流汇聚的地方没有像流敏感分析中一样汇聚，4 个结束节点表示程序中 4 条不同的路径。



(a) 原始 CFG　　(b) 带状态的 CFG　　(c) 扩展的 CFG

图 5-1　原始 CFG、带状态的 CFG 和扩展的 CFG 之间的对比

5.2.1　符号执行框架

由于在扩展的 CFG 上进行后向分析时无法知道程序在结束时的状态信息，而且后向分析不利于状态在分支处的分裂，不符合程序执行的直观感觉，因此符号执行方法是在图 5-1(c)上进行前向分析。其采用的 WorkList 算法框架如下：

```
WL:work list of exploded graph nodes
Get the initial state and construct the first exploded node
Put it into NL
Initialize MaxSteps to a reasonable value,e.g.100000
While (MaxSteps>0 && WL has nodes){
MaxSteps--
N=get node from ML
Process N according to the kind of the program point
Apply the transfer function to generate new nodes and put them into the WL
}
```

算法具体执行步骤如下。

1）在函数的入口处初始化状态

对每个函数参数和全局变量，将其对应的值初始化为符号值。将节点〈函数入口，初始状态〉加入工作列表。在函数体中一般包含常规语句、复合条件语句（&&，||）、选择语句、while 语句、do-while 语句、for 语句等。一般基本块会包含一个结束语句，但是也有可能不包含结束语句，即所有语句都是常规语句。

2）取出节点并处理

从 work list 中取出一个扩展图节点，根据取出节点所在的程序点类型分别进行处理。

（1）Block Edge 节点。

Block Edge 节点是在两个基本块之间的一个点，记录了刚刚处理完的一个块和将要进入的一个块。第一个 Block Edge 节点是一个从 entry 到第一个基本块的节点。之后，在每两个基本块之间都会碰到一个 Block Edge 节点。对 Block Edge 节点的处理很简单，因为没有状态的变化。最简单的处理就是取出目标基本块 B，生成进入 B 的 Block Entrance 节点。不过，为了防止路径出现无限循环，需要对每个基本块进入的次数做记录。在生成 Block Entrance 节点前，检查基本块的访问次数。如果过多，则不生成相应的 Block Entrance 节点，这相当于人为地对状态空间进行了剪枝。

对于 while 和 do-while 循环，一般很难静态地得知确切的循环上界，只能用人为规定最大循环次数的方法。对于 for 循环，一般循环上界是已知的，这时可以从它的常数循环上界中得到一些提示，帮助确定循环次数。但是这也有很多复杂性，如果迭代变量在循环体中被改变了，则不容易得到循环的次数。

事实上，对程序中循环的处理一直是程序分析领域的难题。最好的方法是能够得到一个合适的循环不变量。但是对循环不变量的自动提取仍然是一个未解决的问题。现在一般的做法是规定最大的循环次数，如果超出，就停止对循环体的执行。这种策略对于检查 source-sink 一类的错误相当有效。但是对于 buffer overflow 这类错误效果不好，尤其是当数组下标由循环控制时，很难预先知道循环次数是否会造成数组访问越界，尤其是在循环嵌套出现时，对循环次数的估计更加困难。

（2）Block Entrance 节点。

Block Entrance 节点是在刚刚进入一个块的点，还没有处理该块的第一条语句。对 Block Entrance 节点的处理非常简单，只需要做一些机械的计数器增加工作，并处理一些特殊情况（如空的基本块）。

（3）PostStmt 节点。

PostStmt 节点是最多的一种程序点，在基本块的每条非结束语句的后面都会有这样一个点。对于 PostStmt 节点，访问这个程序点处的下一条语。下一条语句有普通语句和结束语句两种情况，需要分别处理。

① 普通语句。对于普通语句的处理是整个分析中最复杂的一块。需要建立从编程语言语义到定义的模型操作对应。关键的步骤是根据程序语义规则计算出每个表达式的值。得到了表达式的值，就可以根据程序的语义规则及数学规则对程序进行符号计算。

在处理语句的同时，可以检查一些不需要人工描述的程序错误，也可以检查程序员加

入的 assert()语句。

② 结束语句。结束语句是一个分支语句，有两个或更多的后继基本块。结束语句一般会包含一个条件表达式，条件的真或假决定程序执行会选择哪个分支，这个条件被称为路径条件。

3）状态分裂

在分析的过程中，状态的数量会不断增加。由一个状态可能繁衍出多个新的状态，主要有两类。

（1）根据结束语句所包含的路径条件的真假值不同得到两个新的状态：一个对应该结束语句包含的条件为真的情况；另一个对应该结束语句包含的条件为假的情况。因此，需要生成两个后继的 Block Edge 节点，表示状态在此分裂，并沿着两条路径分别继续执行，其算法框架如下：

```
WL:the work list containing all non-ending nodes
B.current block
Terminator:the terminator statement of the block
PC: the set of path condition already collected along the execution path
TrueBlock:the block to be taken if the condition in Terminator is true
FalseBlock:the block to be taken if the condition in Terminator is false
C=get condition from Terminator
if(PC and C is satisfiable)
add C to PC
generate BlockEdge(B,TrueBlock) and put it into WL
if(PC and (not C) is satisfiable)
add (not C) to PC
generate BlockEdge(B,FalseBlock) and put it into WL
```

在上面的算法中，对路径条件做了可满足性判断：if（PC and C is satisfiable）。对路径条件的可满足性判断不是一件容易的事情。在一般情况下，这个问题不可判定。随着研究的不断深入，人们发现了越来越多可判定的理论，并为之设计了判定过程。这方面的工作成果被模块化可满足性理论（Satisfiability Modulo Theory，SMT）汇总到一个统一的框架下，现在已经有了很多成熟的 SMT 求解器可以被用作程序路径条件的判定。

（2）符号值的具体化。在分析的过程中，并不是只有在控制流出现分支的地方状态才会出现分裂。事实上，在任何程序点处都可能出现状态分裂。因为符号执行的变量值是以未知量的形式出现的。在需要已知符号具体值才能继续分析的地方，就必须对符号的每种可能的取值都进行分析，才不会漏掉对可能的状态空间的探索。

在符号分析程序时，有时会碰到无法进行符号执行的情形，而需要进行状态分裂的情况。

① 调用一个行为复杂的库函数，如随机数生成函数。

② 非线性运算，即使以符号的形式执行，交给路径条件求解器也无法求解出来。

③ 某些不想符号执行的运算，如位操作。

④ 循环的边界值是符号值。

⑤ 数组的大小是符号值。

⑥ 数组的下标是符号值。

在这些情形下，为了保证符号执行的效果，需要把符号值具体化，在具体化时，要考查该符号值的取值范围。取值范围包含多种因素，一种是该值固有的范围，若是 char 型值则为－128～127，若是枚举值则在枚举值定义的范围内；另一种因素是先前的路径条件对该符号值的约束，如果将该符号值具体化为一个不符合之前路径条件约束的值，则直接造成路径不可行，也就不用继续分析了。

由于有了这些复杂因素，将符号值进行具体化时也就有了多种方案。

① 最简单的，可以随机生成一个具体值，其他的都不管，一切交给路径条件求解器处理。

② 在取值可能有限的情况下，如数组下标或枚举值，可以穷举所有的可能，将状态分裂。

③ 根据当前已有的路径条件，求解出所有符号值的一组解，用这个解将人们关心的符号值具体化。这样做的优点是可以保证得到一个合法的值，保证继续探索的路径的可行性；缺点也非常明显，即需要大量的计算，降低了分析的效率。

上面几种做法很难简单地判断孰优孰劣，只能根据需要进行选择，如想进行精确而复杂的分析，还是想进行快速、粗略的分析等。

5.2.2 简单例子

本节举一个例子说明符号执行的一些特点，包括路径敏感性、符号值的使用等。考虑下面的代码片段：

```
1 void foo(int n){
2 char *p;
3 if(n!=10)
4 p=(char *) malloc(10);
5 if(n!=10)
6 free(p);
7 }
```

将输入参数 n 设置为符号值。在这段代码中，注意到仅当 n 不等于 10 时，才会有内存分配和释放的行为发生。如果不考虑路径的可行性，那么程序中有 4 条路径：2—3—5—7，2—3—4—5—7，2—3—4—5—6—7，2—3—5—6—7。其中，2—3—5—6—7 会发生释放无效指针的问题，2—3—4—5—7 会发生内存泄漏的问题。但是 2—3—5—6—7 和 2—3—4—5—7 这两条路径实际上是不可行的，即没有 n 的值能同时满足这两条路径的条件：n 不能同时等于 10 又不等于 10。剩下的两条路径 2—3—5—7 和 2—3—4—5—6—7 上的路径条件分别是 n==10 和 n!=10，都可以被满足。而这两条路径上的内存分配和释放操作是匹配的，所以没有错误发生。

在符号执行中，假设 n 是符号值，在探索不同的路径时，收集路径中的约束条件，并进行求解，就可以有效提高分析的精确度。这就是在错误检查中使用路径敏感的符号分析的重要性。如果使用传统的流不敏感数据流分析和流敏感数据流分析，就无法检查出错误或检查出假的错误。