计算机科学
COMPUTER SCIENCE

无尺寸约束的不透明谓词构建算法

王雨芳, 乐德广, Jack TAN, 肖乐, 龚声蓉

引用本文

王雨芳, 乐德广, Jack TAN, 肖乐, 龚声蓉无尺寸约束的不透明谓词构建算法[J]. 计算机科学, 2023, 50(8): 352-358.

WANG Yufang, LE Deguang, Jack TAN, XIAO Le, GONG Shengrong. Opaque Predicate Construction Algorithm Without Size Constraints [J]. Computer Science, 2023, 50(8): 352-358.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

基于多粒度区域相关深度特征学习的行人重识别

Person Re-identification by Region Correlated Deep Feature Learning with Multiple Granularities

计算机科学, 2021, 48(12): 269-277. https://doi.org/10.11896/jsjkx.210400121

进化算法与符号执行结合的程序复杂度分析方法

Program Complexity Analysis Method Combining Evolutionary Algorithm with Symbolic Execution

计算机科学, 2021, 48(12): 107-116. https://doi.org/10.11896/jsjkx.210200052

SymFuzz:一种复杂路径条件下的漏洞检测技术

SymFuzz:Vulnerability Detection Technology Under Complex Path Conditions

计算机科学, 2021, 48(5): 25-31. https://doi.org/10.11896/jsjkx.200600128

基于深度特征融合的图像语义分割

Image Semantic Segmentation Based on Deep Feature Fusion

计算机科学, 2020, 47(2): 126-134. https://doi.org/10.11896/jsjkx.190100119

基于双向KNN排序优化的行人再识别算法

Person Re-identification Algorithm Based on Bidirectional KNN Ranking Optimization

计算机科学, 2019, 46(11): 267-271. https://doi.org/10.11896/jsjkx.181001861

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.220600149

无尺寸约束的不透明谓词构建算法

王雨芳[1,2]　乐德广[2,3]　Jack TAN[3]　肖　乐[2]　龚声蓉[2]
1 苏州大学计算机科学与技术学院　江苏 苏州 215006
2 常熟理工学院计算机科学与工程学院　江苏 苏州 215500
3 威斯康星大学欧克莱尔分校计算机系　威斯康星 欧克莱尔 54701
(3489819475@qq.com)

摘　要　结合不透明谓词,控制流混淆可以进行语义保持的变换,从而达到代码保护的目的。然而,现有的不透明谓词容易遭受符号执行攻击且存在小符号变量问题。为了解决上述问题,结合符号变量和数组利用单数组元素嵌套和符号变量模加运算设计不等条件表达式,并提出无尺寸约束的不透明谓词构建算法。基于该算法构建的不透明谓词混淆可以令攻击者错误地将不透明谓词识别为普通谓词或者将普通谓词识别为不透明谓词,从而有效抵御符号执行攻击。此外,利用不透明谓词检测以及虚假控制流去除等测试程序,对应用了无尺寸约束的不透明谓词混淆后程序的强度、弹性及开销进行实验测试分析。测试结果表明,基于所提算法实现的不透明谓词混淆不仅具有高强度和低开销,而且在新测试环境下仍然具有较高的抗反混淆弹性。

关键词:不透明谓词;符号内存;数组嵌套;代码混淆;符号执行
中图法分类号　TP309

Opaque Predicate Construction Algorithm Without Size Constraints

WANG Yufang[1,2],LE Deguang[2,3],Jack TAN[3],XIAO Le[2] and GONG Shengrong[2]
1 School of Computer Science and Technology,Soochow University,Suzhou,Jiangsu 215006,China
2 School of Computer Science and Engineering,Changshu Institute of Technology,Suzhou,Jiangsu 215500,China
3 Department of Computer Science,University of Wisconsin-Eau Claire,Eau Claire,Wisconsin 54701,USA

Abstract　Combined with opaque predicate,control flow obfuscation enables semantics-preserving transformations,which can achieve the purpose of code protection. However,existing opaque predicate is easily attacked by symbolic execution and has the problem of small symbolic variable. To solve the above problems,combined with symbolic variable and array,this paper designs the conditional expression of inequality by single array nesting and modulo add operation of symbolic variable,based on which an algorithm for constructing opaque predicate without size constraints is proposed. The opaque predicate obfuscation based on the proposed algorithm can incur not only false negative but also false positive issues to attackers,which effectively defends against symbolic execution attacks. Besides,the potency,resilience and cost of the program obfuscated by opaque predicate without size constraints are experimentally tested and analyzed by measuring procedures such as opaque predicate detection,bogus control flow removal and so on. Experimental results show that the opaque predicate obfuscation based on the proposed algorithm not only demonstrates excellent potency and efficient cost,but also has high resilience to anti-deobfuscation in new test environment.

Keywords　Opaque predicate,Symbolic memory,Array nesting,Code obfuscation,Symbolic execution

1　引言

控制流混淆是一种通过更改或复杂化控制流来隐藏程序执行流程的代码混淆技术[1-2],如在程序中插入基于不透明谓词构建的条件判断语句以伪造路径分支并使程序控制流复杂化[3],从而保护真实控制流。其中,不透明谓词是一种其值在混淆前已知,但攻击者很难逆向分析出来的特殊表达式,它包括恒真、恒假和可真可假3种形式[4-5]。因此,求解不透明谓词的难易程度会直接影响控制流混淆的保护效果。

到稿日期:2022-06-16　返修日期:2022-11-16
基金项目:国家自然科学基金(61972059);江苏省产学研合作项目(BY2021280);江苏省自然科学基金(BK20191475);江苏省高校“青蓝工程”中青年学术带头人培养对象项目(2019);江苏省教育科学“十四五”规划课题(C-b/2020/01/29)
This work was supported by the National Natural Science Foundation of China(61972059),Production and Research Cooperation Project of Jiangsu Province(BY2021280),Natural Science Foundation of Jiangsu Province,China(BK20191475),Qing Lan Project of Jiangsu Province in China (2019) and Program of 14th Five Year Plan of Jiangsu Province Education Science(C-b/2020/01/29).
通信作者:乐德广(ledeguang@cslg.edu.cn)

当前，研究人员一般采用数学定理、混沌理论和纠缠量子位等方法构建不透明谓词。Chen 等[6]利用同余方程组解的状态构建不透明谓词，并采用中国剩余定理判断谓词输出。通过复杂的数学变换可以隐藏谓词内部逻辑，使其具有较好的抗静态逆向分析能力，但是在运行过程中此谓词值是固定的。Su 等[7]构建的基于 En_Logistic 映射的混沌不透明谓词具有很高的保密性及谓词结果的不确定性。Xie 等[8]提出基于二维超混沌映射的不透明谓词构建方法，它将谓词求解问题转化为求解超混沌问题以增加条件判断的难度。Su 等[9]采用二维帐篷混沌映射构建不透明表达式并使用此表达式替换不透明谓词中的常量，从而增加谓词破解难度且开销较小。Balachandran[10]使用多对纠缠量子位来创建量子不透明谓词，此不透明谓词的值不能通过静态推导得到，这增加了程序的分支复杂度和随机性。Tung 等[11]基于程序语义中变量值的集合构建不透明谓词，它与普通谓词具有相同的语法及语义特征且可以抵抗启发式攻击与自动攻击。

以上方法只考虑能够引入虚假控制流的不透明谓词，或者只考虑在已有判断语句中通过复杂化谓词表达式来提高混淆程序的复杂度，造成攻击者对不透明谓词检测的漏判，并未考虑通过在现有的条件表达式中构建不透明谓词来引起攻击者对不透明谓词的错误检测。

此外，随着符号执行在路径探索、约束求解和内存建模等方面求解能力的不断增强[12]，现有的不透明谓词混淆也面临符号执行逆向分析的威胁，攻击者利用符号执行使用代表任意输入数据的符号对不透明谓词混淆程序进行分析以实现不透明谓词检测[13]。Ming 等[14]从不透明谓词混淆程序分支中提取约束并将它们组成公式来表示混淆程序的执行逻辑，对约束进行符号求解，识别出相应的不透明谓词，从而正确检测出不透明谓词。

为抵抗符号执行对不透明谓词的检测攻击，Xu 等[15]利用符号内存问题构建引起符号执行攻击漏检和错检的不透明谓词。文献[15]构建出的具有双重属性的不透明谓词弹性较高，但是其符号执行的求解结果依赖于声明的符号变量的大小，即当声明的符号变量较小时，保护效果较弱，因此其存在小符号变量问题。为解决此问题，本文利用符号变量与嵌套变量间的不等价关系使得它们的可能取值不断增长的性质，提出无尺寸约束的不透明谓词，并给出该不透明谓词的构建算法，从而形成高弹性的不透明谓词混淆。

2 小符号变量问题分析

如果将符号变量作为内存的地址值（如指针或偏移量等），并通过它从内存中读取相应的值[16-17]，会使得符号执行难以根据符号变量获取对应的内存值并导致求解能力下降，甚至引发错误，这被称为符号内存问题。文献[15]利用符号变量能够引发符号内存问题这一性质构建不透明谓词，如图 1 所示。

从图 1 可以看出，它定义了双数组 $l1_ary$ 及 $l2_ary$，并将符号变量 j 作为内存的地址值，通过它从内存中读取出相应的数组值以构建变量 i，再构建不透明谓词 $i==j$ 和 $i==1$ && $j==7$，分别称为 I 型不透明谓词和 II 型不透明谓词。

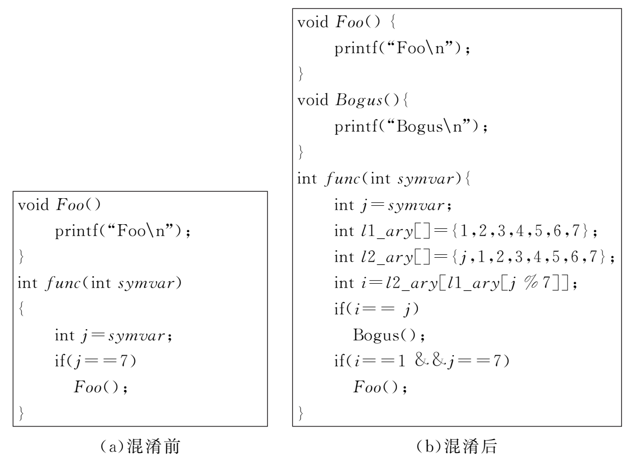


图 1 基于文献[15]的混淆示例

Fig. 1 Example of obfuscated program based on reference [15]

在对以上例子使用 Angr 符号执行引擎进行符号执行分析时发现，当声明 2 字节及以上的符号变量时，符号执行会生成错误的路径约束，令符号执行难以得到满足这个条件的解。其中，符号执行找出一条 Bogus 路径的解（−4）为错解，说明它误把 I 型不透明谓词识别为普通谓词；并且它无法找出一条 Foo 路径的合适的解，即无解，说明它误把 II 型不透明谓词（即普通谓词）识别为不透明谓词。

而声明 1 字节符号变量时，符号执行无法找到执行 *Bogus*() 函数的解，即无解，说明它能正确地识别出 I 型不透明谓词。另外，符号执行也能够找到正确执行 *Foo*() 函数的解(7)，即正解，说明它能正确地识别出 II 型不透明谓词，从而引起小符号变量问题。这是因为声明 1 字节符号变量时，符号变量所代表的数值范围有限，无符号数值范围为 0～255，有符号数值范围为 −128～127。模拟程序执行时，符号执行利用执行树来表示程序的所有执行路径，并且在给定的运行时间内遍历执行树以探索出所有依赖于符号变量的执行路径。数值范围的有限使得符号执行易于获取符号变量的实际值即生成正确的解，并获知触发此解的执行路径，这意味着不透明谓词失效。

此外，由于符号执行在应对路径爆炸、约束求解和内存建模等方面的挑战时性能不断优化，如 Angr 符号执行引擎使用模拟管理器替换路径组，它在模拟执行过程中保存控制流状态而非控制流路径，这使得文献[15]混淆算法构建的不透明谓词的真实取值易于被符号执行求解，从而导致其在设置 2 字节和 4 字节符号变量的情况下保护效果减弱。

为解决小符号变量问题，并降低基于路径探索和约束求解优化的不透明谓词检测攻击的有效性，本文利用符号变量和数组在基于单数组元素嵌套和符号变量模加运算的基础上构建无尺寸约束的不透明谓词，下面给出其构建算法。

3 基于无尺寸约束的不透明谓词构建算法

本文算法针对符号执行的符号内存困难问题，首先利用数组以及符号变量进行单数组嵌套运算，并将运算结果赋予一变量，称其为嵌套变量。基于嵌套变量与符号变量模加

运算的不等价关系构建的恒假表达式即为不透明谓词 I。此外，针对待保护的条件表达式，构建关于嵌套变量和符号变量的恒真表达式，将此表达式通过与逻辑运算引入待保护的条件表达式中形成不透明谓词 II，从而构成高弹性的无尺寸约束的不透明谓词。其中，无尺寸约束指符号执行求解结果不依赖于声明的符号变量的大小，即声明 1 字节符号变量时符号执行难以求解出符合条件的解。其具体构建如算法 1 所示。

算法 1 无尺寸约束的不透明谓词构建算法

输入：待保护函数 func(parameters)

输出：I 型不透明谓词 Op_I，II 型不透明谓词 Op_II

1. 遍历待保护函数 func(parameters)，得到所有的 if 条件 conditions；
2. 对于∀ conditions，∃ int if_int_vars，则有 cond(if_int_vars)；
3. 利用 cond(if_int_vars)和 func(parameters)中的公共变量构建符号变量 sym_var={if_int_vars ∩ parameters}，对应待保护条件表达式为 cond (sym_var)；
4. 定义一个元素初始值从 0 开始，步长为 1 的递增正整数数组 a[10]={0,1,…,8,9}；
5. 根据步骤 3 和步骤 4，基于单数组嵌套运算构建嵌套变量 nest_var=a[a[sym_var%9+1]]；
6. 基于嵌套变量和符号变量模加运算构建恒假不等条件表达式 nest_var!=sym_var%9+1，则 I 型不透明谓词 Op_I 为 nest_var!=sym_var % 9+1；
7. 构建恒真的不等条件表达式 nest_var ! = sym_var % 9，再将该表达式与 cond(sym_var)进行与逻辑运算得 nest_var!= sym_var % 9 &&cond(sym_var)，则 II 型不透明谓词 Op_II 为 nest_var!=sym_var % 9 &&cond(sym_var)。

在算法 1 中，首先从输入的待保护函数 *func*(*parameters*)中遍历所有的 if 条件 *conditions*，当 *conditions*==∅ 时，说明此函数不符合要求，否则从中筛选出所有的整型变量 *if_int_vars*。如果 *if_int_vars*≠∅ ∩ *if_int_vars* −>*getType*() −>*isIntegerTy*()==*True*，则说明 if 条件中存在整型变量，同时得到对应的条件表达式 *cond*(*if_int_vars*)。其次，通过 *cond*(*if_int_vars*)中的 *if_int_vars* 和 *func*(*parameters*)中的 *parameters* 求其公共变量{*if_int_vars* ∩ *parameters*}，并利用此公共变量来构建符号变量 *sym_var*，即 *sym_var*={*if_int_vars* ∩ *parameters*}，此变量所对应的条件表达式即为待保护条件表达式 *cond* (*sym_var*)。然后，使用静态方式定义值均为正整数的数组 *a*[10]={0,1,…,9}，并结合符号变量 *sym_var* 基于单数组嵌套运算构建嵌套变量*nest_var*=*a*[*a*[*sym_var*%9+1]]。根据嵌套变量与符号变量间的不等价关系所构建的恒假表达式 *nest_var* !=*sym_var* % 9+1 即为 I 型不透明谓词 Op_I。最后，构建关于符号变量与嵌套变量的恒真表达式 *nest_var* != *sym_var* % 9，将此表达式与待保护条件表达式 *cond*(*sym_var*)进行与操作所得到的表达式 *nest_var* != *sym_var* % 9 &&*cond*(*sym_var*)即为 II 型不透明谓词 Op_II。

图 2 给出了一个示例，它显示了在未混淆的代码中，基于上述算法构建的不透明谓词可以实现控制流混淆。

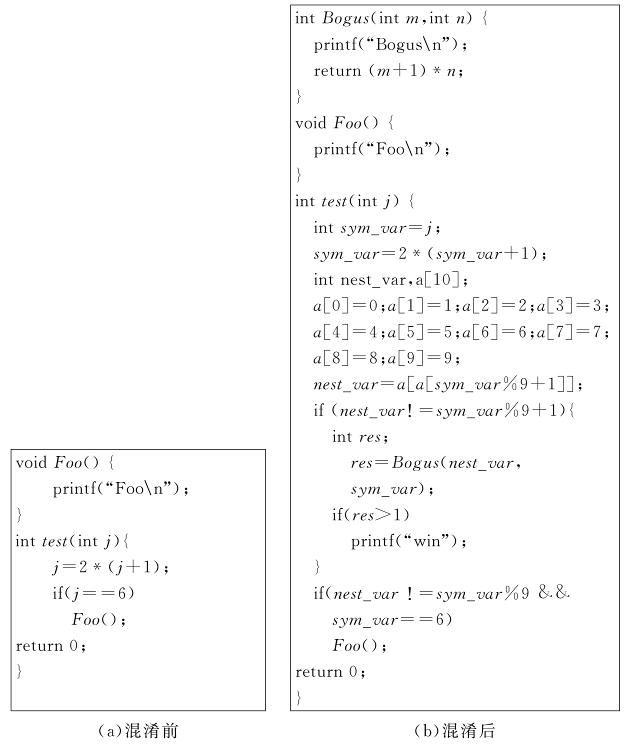


图 2 基于本文算法的混淆示例

Fig. 2 Example of obfuscated program based on the proposed algorithm

从图 2 可以看出，混淆前的受保护函数 *test*(*int j*)接收一个形式参数 *j*。首先遍历 *test* (*int j*)函数得到所有的 if 条件，此函数只包含一个 if 条件即 *if*(*j*==6)，该 if 条件仅有一个整型变量 *j*，其对应的条件表达式 *j*==6 即为待保护表达式。由于待保护条件表达式中的变量 *j* 与 *test*(*int j*)函数中的参数 *j* 为公共变量，利用此公共变量构建符号变量即 *sym_var*=*j*，同时需要保证待保护条件表达式的语义不变，将其表示为 *sym_var*==6 且表达式 *j*=2*(*j*+1)被表示为 *sym_var*=2*(*sym_var*+1)。其次，使用静态方式定义数组 *a*[10]={0,1,…,9}及嵌套变量 *nest_var*=*a*[*a*[*sym_var* % 9+1]]。然后，构建 I 型不透明谓词 *nest_var* != *sym_var* % 9+1，同时将嵌套变量及符号变量传入 *Bogus*()函数进行(*m*+1)***n* 计算并将计算结果返回。接着，根据返回值确定是否打印字符串"*win*"，以此作为虚假控制流。最后，构建恒真的表达式 *nest_var*! =*sym_var*%9，并将其与待保护条件表达式 *sym_var*==6 进行与操作以构建 II 型不透明谓词 *nest_var*! =*sym_var*%9 &&*sym_var*==6。

基于上述算法构建混淆函数的控制流图，同时与原函数的控制流图进行对比分析，具体如图 3 所示。从图 3(b)的混淆函数控制流图可以看出，I 型不透明谓词对应的条件 if(*nest_var*!= *sym_var* % 9+1)为恒假条件，它永远不可能满足，如果符号执行引擎缺乏相应处理机制来解决符号内存问题，它生成的错误约束将导致其将不透明谓词识别为普通谓词并保留虚假控制流。此外，当 *test*(*int j*)函数的实参为 2 时，符号变量 *sym_var* 取值为 6，*nest_var* != *sym_var* % 9 不会影响待保护条件表达式 *sym_var*==6 中 *sym_var* 的取值，并且

只有 $nest_var \ != sym_var \% 9$ 为真时 $sym_var == 6$ 才会被运算。如果符号执行引擎不支持符号内存，那么它就难以处理 $nest_var \ != sym_var \% 9$ 的约束，也不能到达 $sym_var == 6$，从而错误地将普通谓词识别为不透明谓词并删除正常控制流。

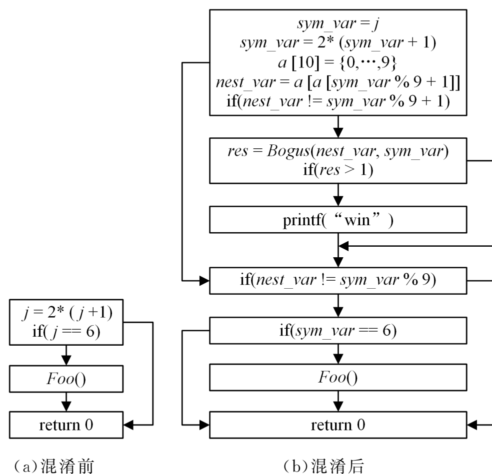


图 3 基于本文算法的混淆函数控制流图

Fig. 3 Control flow graph of obfuscated function based on the proposed algorithm

4 实验测试与评估

为了测试本文算法的效果，本节分别使用 Tigress 中的不透明谓词构造算法[18]、文献[15]算法以及本文算法来构建不透明谓词并将其插入待保护程序中进行控制流混淆，然后从强度、弹性及开销[4]等方面进行具体测试和比较，并从数组、嵌套变量和条件表达式的构建方式以及控制流去除的有效性等方面进行详细评估。其中，强度指理解混淆程序的难易程度，混淆程序越难理解，则其强度越高。弹性用于评估混淆技术如何抵抗自动化反混淆攻击，本文利用符号执行攻击以评估不透明谓词的安全性。开销指混淆程序所产生的额外资源消耗，包括时间或空间开销。实验服务器环境为操作系统 Ubuntu 16.04.7 LTS，2×14 核处理器 Inter(R) Xeon(R) Gold 6132 @ 2.604GHz，503 GB 内存和 9 TB 硬盘。

4.1 强度测试

在斐波那契数列的递归求解程序[19]中，使用文献[18]算法、文献[15]算法以及本文算法构建的不透明谓词来引入虚假分支路径，以增加程序控制流的复杂程度。首先使用程序长度[20]、控制流循环复杂度[21]和数据流复杂度[22]指标来度量混淆前后程序的强度。其中，程序长度 N 指程序中操作符 N_1 和操作数 N_2 的个数，如式(1)所示：

$$N = N_1 + N_2 \tag{1}$$

控制流循环复杂度 $V(G)$ 指程序中路径的数量，如式(2)所示：

$$V(G) = e - n + 2 \tag{2}$$

其中，e 表示控制流图中边的数量，n 表示控制流图中结点的数量。

数据流复杂度 DFC 指数据流图中数据边的数量 $count$，如式(3)所示：

$$DFC = count \tag{3}$$

测试结果如表 1 所列。

表 1 强度测试结果

Table 1 Test results of potency

测试用例	程序长度	控制流循环复杂度	数据流复杂度
原程序	95	4	6
文献[18]算法	200	5	10
文献[15]算法	234	6	12
本文算法	305	7	14

从表 1 可以看出，各程序长度都大幅增加，本文算法的增加幅度高于文献[18]及文献[15]算法，达到了原程序的 3 倍。文献[18]算法、文献[15]算法与本文算法的控制流循环复杂度分别增加到原程序的 1.25 倍、1.5 倍和 1.75 倍，且数据流循环复杂度分别增加到原程序的 1.67 倍、2 倍和 2.33 倍。为了进一步了解斐波那契数列递归求解程序混淆前后控制流分支的变化情况，使用 SourceMoniter[23] 分别从语句数、分支语句比例和最大嵌套深度 3 个指标进行测试与分析，其中语句数表示函数中语句的个数，分支语句比例表示分支语句数占语句数的比例，最大嵌套深度表示分支嵌套的最大层数。测试结果如表 2 所列。

表 2 控制流分支复杂度测试结果

Table 2 Test results of control flow branch complexity

测试用例	语句数	分支语句比例/%	最大嵌套深度
原程序	24	16.7	2
文献[18]算法	815	1.8	6
文献[15]算法	32	15.6	2
本文算法	45	13.3	3

从表 2 可以看出，混淆程序的语句数相比原程序均有所增加，本文算法的增加幅度为 1.875 倍，高于文献[15]算法的 1.33 倍，但都远小于文献[18]算法的语句数。语句数与分支语句比例的乘积为分支语句数，计算得到原程序、文献[18]算法、文献[15]算法与本文算法的分支语句数分别为 4，15，5 和 6。文献[18]算法增加的分支条件主要是由 Tigress 在对应的混淆程序中插入了 goto 分支语句以及流操作函数等内容所引起。文献[15]算法与本文算法增加的 1 个分支条件即为 I 型不透明谓词所在的条件。由于本文算法在 I 型不透明谓词对应条件内部额外添加了一个 if 分支条件，因此本文算法的分支语句数为 6。同时，在原程序与文献[15]算法的最大嵌套深度均为 2 的情况下，额外添加的分支条件使得本文算法的最大嵌套深度增加了 1，即为 3，而文献[18]算法的最大嵌套深度增加了 4，即为 6。因此，从 SourceMoniter 测试结果来看，混淆后的程序比原程序更加复杂。

4.2 弹性测试

使用基于符号执行的程序分析技术作为反混淆的核心，通过它来测试不透明谓词的抗检测弹性。由于 Angr 符号执行引擎免费开源且在路径探索及约束求解方面的效果较好[24]，因此选择它进行弹性测试。如果它求解不透明谓词时产生漏检和错检，则说明此谓词的弹性较高，否则说明其弹性较低。首先，安装基于 Python 3.5 的虚拟环境 virtualenvwrapper 4.8.4，并使用 pip3 安装 angr 8.19.10.30。然后，使用不透明谓词检测程序 OPDetection.py 对基于不透明

谓词的混淆程序进行 Angr 求解，OPDetection. py 的关键代码如图 4 所示。

```
path_to_binary="name of program"
project = angr.Project(path_to_binary)
sym_var=claripy.BVS("sym_val",8 * length)
ini_state= project.factory.entry_state(args=[project.filename,sym_var ])
simulation = project.factory.simgr(ini_state)
def is_successful(state):
  #查找函数地址，可达则返回 true
simulation.explore(find=is_successful)
if simulation.found:
  #打印出函数结果
else:
  #报告未找到
```

图 4 Angr 脚本 OPDetection. py

Fig. 4 Angr script OPDetection. py

其中，*is_successful*()函数会将输入的 I 型不透明谓词或者 II 型不透明谓词的地址作为目标地址，再使用 *explore*()函数求解出符合条件的解。Angr 的求解结果分为 3 种情况：无解、错解和正解。其中，针对 I 型不透明谓词，如果为无解则表示 Angr 无法求解出满足条件的解并将此谓词识别为不透明谓词；如果为错解则表示 Angr 求解出的解根本不可能满足恒假的条件并将此谓词识别为普通谓词；如果为正解则表示 Angr 可以求解出满足条件的解并将此谓词看作普通谓词。针对 II 型不透明谓词，如果为无解则表示 Angr 将此谓词识别为不透明谓词并将此谓词所在分支看作虚假分支；如果为错解则表示 Angr 求解出的解实际上并不满足条件但 Angr 却认为其满足条件，从而将此谓词看作普通谓词；如果为正解表示 Angr 求解正确且满足条件，从而将此谓词识别为普通谓词。因此，如果 Angr 求解 I 型不透明谓词的结果为错解且求解 II 型不透明谓词的结果为无解，说明此不透明谓词弹性较高。

设置 1～4 字节的符号变量，使用程序 OPDetection. py 在本文测试环境下使用文献[18]算法、文献[15]算法和本文算法的混淆程序分别进行 Angr 求解，并与文献[15]的测试结果进行比较。相较于文献[15]测试环境使用路径组来保存控制流路径再进行求解，本文测试环境先使用模拟管理器来保存控制流状态再进行求解。具体结果如表 3 所列。

表 3 弹性测试结果

Table 3 Test results of resilience

	文献[15]测试环境		本文测试环境					
	文献[15]算法		文献[18]算法		文献[15]算法		本文算法	
符号变量长度	I 型	II 型	I 型	II 型	I 型	II 型	I 型	II 型
1 字节	无解	正解	无解	正解	无解	正解	错解	无解
2 字节	错解	无解	无解	正解	无解	正解	错解	无解
3 字节	—	—	无解	正解	无解	正解	错解	无解
4 字节	错解	无解	无解	正解	无解	正解	错解	无解

其中，—表示未测试，I 型及 II 型分别表示 I 型不透明谓词及 II 型不透明谓词，文献[18]算法无法保护 II 型不透明谓词则其求解结果恒为正解。从表 3 可以看出，文献[15]的算法不仅存在小符号变量问题，而且其与文献[18]算法在本文的 angr 8.19.10.30 测试环境下设置 1,2,3 及 4 字节的符号变量时，I 型不透明谓词的求解结果都为无解，说明 Angr 将此谓词识别为不透明谓词，II 型不透明谓词的求解结果都为正解，说明 Angr 能得到此谓词的正确取值并将其识别为普通谓词，从而说明其不会错判和漏判。

而在本文的 angr 8.19.10.30 测试环境下求解基于本文算法的混淆程序时，符号变量长度在[1-4]字节范围内，不透明谓词检测程序求解 I 型不透明谓词的结果都为错解，说明其将此不透明谓词识别为普通谓词，求解 II 型不透明谓词的结果都为无解说明其将此普通谓词识别为不透明谓词，且不存在小符号变量问题。

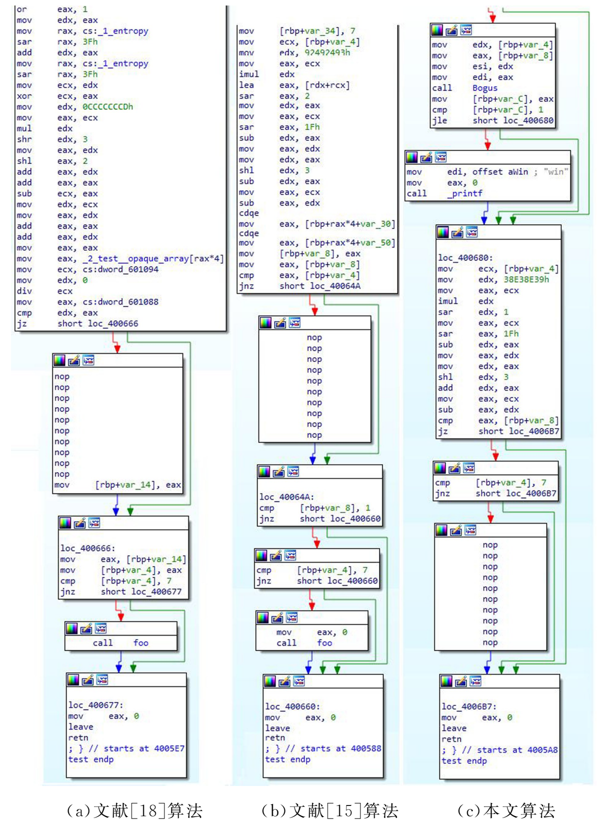
在本文测试环境下，进一步利用虚假控制流去除程序 Debogus. py[25] 对混淆程序进行虚假控制流去除处理，结果如表 4 所列。

表 4 虚假控制流去除结果

Table 4 Results of bogus control flow removal

	文献[18]算法		文献[15]算法		本文算法	
符号变量长度	I 型	II 型	I 型	II 型	I 型	II 型
1 字节	去除	保留	去除	保留	保留	去除
2 字节	去除	保留	去除	保留	保留	去除
3 字节	去除	保留	去除	保留	保留	去除
4 字节	去除	保留	去除	保留	保留	去除

根据表 4 中的结果，以 1 字节的符号变量为例，对于利用 Debogus. py 去除虚假控制流后的二进制程序，使用 IDA 7.2[26] 构建反混淆后的程序控制流，如图 5 所示。



(a)文献[18]算法 (b)文献[15]算法 (c)本文算法

图 5 虚假控制流去除处理后的控制流图

Fig. 5 Control flow chart after bogus control removal

在图 5(a)中，文献[18]算法添加的虚假条件会随机添加某个函数的调用语句，Debogus. py 成功将此语句替换为 nop 指令并保留 *Foo*()函数的调用语句。从图 5(b)可以看出，*Bogus*()函数的调用语句被替换为 nop 指令而 *Foo*()函数的

调用语句仍保留，说明 Debogus. py 成功将文献[15]中构建的虚假谓词去除并保留真实的谓词。而针对使用本文算法构建的不透明谓词，*Bogus*()函数的调用语句仍保留而 *Foo*()函数的调用语句被替换为 nop 指令，如图 5(c)所示，这说明 Debogus. py 错误地将不透明谓词识别为普通谓词并造成漏判，且将普通谓词识别为不透明谓词并造成错判。

结合表 3、表 4 及图 5 可以看出，针对文献[18]和文献[15]算法构建的不透明谓词，不透明谓词检测程序 OPDetection. py 的求解结果分别为无解和正解。虚假控制流去除程序 Debogus. py 去除该混淆程序中的 I 型不透明谓词并保留 II 型不透明谓词。针对本文算法构建的不透明谓词，不透明谓词检测程序 OPDetection. py 求解结果分别为错解和无解。虚假控制流去除程序 Debogus. py 保留该混淆程序中的 I 型不透明谓词并去除 II 型不透明谓词。本文算法不仅有效克服了小符号变量问题，而且在 angr 8.19.10.30 的符号执行新环境中仍具有较高的抗反混淆弹性。

4.3 开销测试

开销包括运行时间和存储开销两部分。其中，运行时间指程序的执行时间，存储开销指程序所占的磁盘空间，可以用程序大小来衡量。使用文献[18]算法、文献[15]算法及本文算法混淆斐波那契数列的递归求解程序[19]，分别求解出斐波那契数列前 10 项的值，并多次测量混淆前后程序的运行时间，测试结果如图 6 所示。

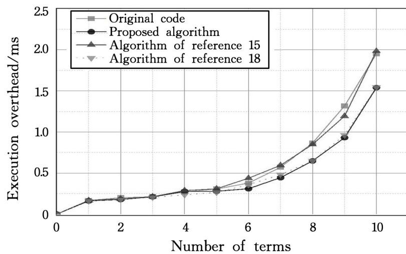


图 6 斐波那契数列求解程序的运行时间

Fig. 6 Execution overhead of Fibonacci solver program

从图 6 可以看出，求解斐波那契数列前 10 项时，混淆前后程序的运行时间随项数的增加呈指数级增长，使用文献[15]算法混淆程序的运行时间与原程序的差距较小，文献[18]算法混淆程序的运行时间与本文算法的差距较小。第 6 项之后，使用本文算法混淆程序的运行时间比文献[15]算法混淆程序短，说明本文算法构建的不透明谓词不仅不影响程序的运行时间，而且随着程序运算量增加，其时间开销小于文献[15]算法。同时，除了第 3 项、第 4 项以及第 5 项，在其他项上文献[18]的算法混淆程序的运行时间均比本文算法长。进一步计算递归求解斐波那契数列前 10 项的总平均执行时间并测量混淆前后的程序大小，测试结果如表 5 所列。其中，平均运行时为递归求解斐波那契数列前 10 项的总平均执行时间。从表 5 可以看出，本文算法混淆后程序的总体平均运行时间均比原程序、文献[18]与文献[15]算法短，说明本文算法的总体时间开销性能更好。在混淆前后程序大小变化方面，混淆后的程序大小增加幅度不大。使用文献[18]算法的混淆程序大小为原程序的 1.24 倍；使用文献[15]算法的混淆程序大小为原程序的 1.045 倍；使用本文算法的混淆程序大小为原程序的 1.053 倍，说明其存储开销不会对资源存储造成负担。综合运行时间和程序所占磁盘空间这两种度量指标来看，基于本文算法的混淆程序开销较小。

表 5 开销指标测试结果

Table 5 Test results of costs

测试用例	平均运行时/ms	程序大小/kB
原程序	0.6238	7.10
文献[18]算法	0.5040	8.82
文献[15]算法	0.6219	7.42
本文算法	0.4979	7.48

4.4 算法评估

最后，分别从是否使用静态方式构建数组、是否使用数组嵌套方式构建嵌套变量、是否保护等价判断条件、谓词 I 是否构建嵌套变量与符号变量的不等价关系、谓词 II 是否引入恒真条件表达式、能否解决小符号变量问题、是否考虑 3 字节的符号变量，以及能否抵抗虚假控制流去除这几个方面对文献[18]算法、文献[15]算法与本文算法进行评估，结果如表 6 所列。

由表 6 可知，文献[18]算法、文献[15]算法与本文算法均使用静态方式构建数组，且文献[15]算法及本文算法均对等价判断条件进行保护。虽然后两种混淆算法均通过数组嵌套的方式来构建嵌套变量，但文献[15]算法是使用双数组嵌套的方式构建嵌套变量，而本文算法使用单数组嵌套的方式构建。此外，文献[18]算法通过 AddOpaque 变换来构建谓词 I 的等价关系，但其无法保护 II 型不透明谓词。同时，文献[15]算法利用嵌套变量与符号变量的等价关系构建谓词 I，并基于嵌套变量的取值构建谓词 II，而本文算法利用嵌套变量与符号变量的不等价关系构建谓词 I，并使用基于嵌套变量与符号变量的恒真表达式构建谓词 II，因此，文献[18]与文献[15]难以解决小符号变量问题。而本文算法不仅能解决此问题，还考虑到 3 字节的符号变量，且能够更好地抵抗虚假控制流去除处理。

表 6 算法评估

Table 6 Algorithms evaluation

评估指标	文献[18]算法	文献[15]算法	本文算法
使用静态方式构建数组	√	√	√
使用数组嵌套方式构建嵌套变量	×	√	√
是否保护等价判断条件	×	√	√
谓词 I 是否构建嵌套变量与符号变量的不等价关系	×	×	√
谓词 II 是否引入恒真条件表达式	×	×	√
能否解决小符号变量问题	×	×	√
是否考虑 3 字节的符号变量	×	×	√
能否抵抗虚假控制流去除	×	×	√

结束语 随着符号执行技术在路径探索和约束模型求解方面的发展，它对不透明谓词的保护效果造成了极大威胁，从而未能误导符号执行工具实现错判与漏判。本文针对符号执行的弱点及小符号变量问题，从符号内存这一问题出发，研究了无尺寸约束的不透明谓词并对相应的构建算法进行了详细的描述。本文算法使用单数组嵌套方式构建的无尺寸约束的不透明谓词不仅克服了小符号变量问题，而且大大增加了符号执行的难度。同时，利用本文算法混淆待保护函数能增加其程序长度、控制流循环复杂度和数据流复杂度，且几乎

不增加运行时间和程序大小。因此,本文算法能够成功隐藏程序的控制流信息并利用虚假条件来误导攻击者,具有较好的保护效果。为了进一步增加不透明谓词混淆的安全性及多样性,扩展数组与嵌套变量的构建方式(如使用动态方式定义数组或者使用逻辑运算方式构建嵌套变量),以及多样化符号变量的输入方式(如使用标准输入或者用户命令输入),是下一步研究的方向。

参 考 文 献

[1] SCHRITTWIESER S, KATZENBEISSER S, KINDER J, et al. Protecting software through obfuscation: Can it keep pace with progress in code analysis? [J]. ACM Computing Surveys, 2016, 49(1): 1-37.

[2] HOSSEINZADEH S, RAUTI S, LAUREN S, et al. Diversification and obfuscation techniques for software security: A systematic literature review [J]. Information and Software Technology, 2018, 104(5): 72-93.

[3] XU H, ZHOU Y F, MING J, et al. Layered obfuscation: A taxonomy of software obfuscation techniques for layered security [J]. Cybersecurity, 2021, 9(3): 1-18.

[4] COLLBERG C, THOM BORSON C D, DOUGLAS L. A taxonomy of obfuscating transformations [R]. Auckland: Department of Computer Science, University of Auckland, 1997.

[5] COLLBERG C, THOMBORSON C D, DOUGLAS L. Manufacturing cheap, resilient, and stealthy opaque constructs [C] // Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98). New York, NY, USA: ACM, 1998: 184-196.

[6] CHEN D M, FAN X H, ZHU J, et al. Obfuscation algorithms based on congruence equation and Chinese remainder theorem [J]. Application Research of Computers, 2015, 32(2): 485-488.

[7] SU Q, WU W M, ZHANG Z L, et al. Research and application of chaos opaque predicate in code obfuscation [J]. Computer Science, 2013, 40(6): 155-159.

[8] XIE X, LIU F L, LU B, et al. Mixed obfuscation of overlapping instruction and self-modify code based on hyper-chaotic opaque predicates [C] // Proceedings of 2014 Tenth International Conference on Computational Intelligence and Security. New York, NY, USA: ACM, 2014: 524-528.

[9] SU Q, SUN J T. Research on opaque predicate obfuscation technique based on chaotic opaque expression [J]. Computer Science, 2017, 44(12): 114-119.

[10] BALACHANDRAN V. Quantum obfuscation: Quantum predicates with entangled qubits [C] // Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy (CODASPY '21). New York, NY, USA: ACM, 2021: 293-295.

[11] TUNG Y J, HARRIS I G. Zero footprint opaque predicates: Synthesizing opaque predicates fromnaturally occurring invariants [C] // Proceedings of the 2021 International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Berlin: Springer, 2021: 299-318.

[12] ZHANG Y F, CHEN Z B, SHUAI Z Q, et al. Multiplex symbolic execution: exploring multiple paths by solving once [C] // Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. New York, NY, USA: ACM, 2020: 846-857.

[13] BALDONI R, COPPA E, CONO D D, et al. A survey of symbolic execution techniques [J]. ACM Computing Surveys, 2018, 51(3): 1-39.

[14] MING J, XU D P, WANGL, et al. LOOP: Logic-oriented opaque predicate detection in obfuscated binary code [C] // Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. New York, NY, USA: ACM, 2015: 757-768.

[15] XU H, ZHOU Y F, KANG Y, et al. Manufacturing resilient bi-opaque predicates against symbolic execution [C] // Proceedings of 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). New York, NY, USA: IEEE, 2018: 666-677.

[16] LIANG H L, YU W Q, AI L, et al. A practical concolic execution technique for large scale software systems [C] // Proceedings of the Evaluation and Assessment in Software Engineering (EASE'20). New York, NY, USA: ACM, 2020: 312-317.

[17] XU H. Software obfuscation with layered security [D]. Hong Kong: The Chinese University of Hong Kong, 2018.

[18] UNIVERSITY OF ARIZONA. Tigress software [EB/OL]. https://tigress.wtf/addOpaque.html.

[19] RUNOOB. Fibonacciprogram [EB/OL]. https://www.runoob.com/cprogramming/c-examples-fibonacci-series.html.

[20] MUSLIJA A, ENOIU E. On the measurement of software complexity for PLC industrial control systems using TIQVA [C] // Proceedings of the 35th Annual ACM Symposium on Applied Computing. New York, NY, USA: ACM, 2020: 1556-1565.

[21] ZHAO Y J, TANG Z Y, WANG N, et al. Evaluation of code obfuscating transformation [J]. Journal of Software, 2012, 23(3): 700-711.

[22] MENST. Research trends in structural software complexity [EB/OL]. https://arxiv.org/abs/1608.01533v1.

[23] CAMPWOOD. Source Moniter software [EB/OL]. https://www.campwoodsw.com/sourcemonitor.html.

[24] ANGR. Angr software [EB/OL]. http://angr.io/.

[25] BLUESADI. Debogus program [EB/OL]. https://github.com/bluesadi/debogus.

[26] HEX RAYS. IDA Pro [EB/OL]. https://hex-rays.com/IDA-pro/.

WANG Yufang, born in 1997, postgraduate. Her main research interests include information security and so on.

LE Deguang, born in 1975, Ph.D, associate professor. His main research interests include information security and cryptography.

(责任编辑:何杨)