击败 SOTA 反混淆方法

程瑞[1], 黄泳龙[1], 徐辉[2], and 范铭[3]

[1] 西安交通大学，软件学院
[3] 西安交通大学，网安学院
[2] 复旦大学，计算机学院

摘要

代码混淆是一种被广泛应用的软件保护技术。目前，有许多基于符号执行、污点分析、切片等软件分析方法的自动化反混淆方法被提出，传统代码混淆方法正面临极大的挑战。同时，也有许多针对符号执行的混淆方法被提出。这些混淆方法大部分是利用路径爆炸这一挑战性问题提出的，然而许多反混淆方法并不会使用符号执行工具去覆盖程序的所有可能执行路径，而仅仅是尝试覆盖所有基本块，以重建控制流图。还有一些路径合并策略（如 veritesting）能有效缓解动态符号执行中存在的路径爆炸问题，而研究者们都没有验证他们提出的混淆方法能否有效抵抗这些策略。基于这些，我们合理地提出第 1 个问题：已提出的反符号执行的代码混淆方法真的能有效抵抗自动化的反混淆方法吗？部分混淆方法是通过调用系统 API 实现的，但是，为符号执行工具编写扩展，处理 API 语义并不是复杂的工作。基于这些，我们提出第 2 个问题：是否为符号执行工具编写简单的扩展就能导致这些混淆方法失效？为了回答这两个问题，我们深入研究了已有的反符号执行混淆对自动化反混淆方法的抵抗性。结果显示，现有的反符号执行混淆方法可以分为 3 类：(1) 对自动化反混淆方法并没有明显的抵抗性；(2) 对自动化反混淆方法有明显的抵抗性，但通过为符号执行工具编写简单的扩展，自动化反混淆方法就可以处理它们；(3) 有明显的抵抗性，并且很难通过扩展符号执行工具处理它们。我们深入研究了第 3 类混淆，提出了构造这一类混淆的框架。基于我们提出的框架，软件开发者们可以简单地构造出高强度的混淆方法。实验显示，基于我们的框架构造出的混淆方法具有更小的时间开销。我们还提出了一种引入过污染问题的混淆构造框架，它也能让切片工具错误地选择不相关代码。我们基于 LLVM 编译工具链实现了一个混淆编译器，它能自动化输出被混淆过的可执行文件。本文的所有实验与工具的源代码已开源至 Github: https://github.com/DrShabulaji/Paper_Obfs.

1 引言

代码混淆 [7] 是一种被广泛应用于软件保护的技术。可以将代码混淆定义为一个转换过程 O，它的输入是原始程序 P，输出是混淆后的程序 P'，即 O(P) = P'，同时 P' 还应该具备两条性质：(1) P' 与 P 的语义应该是相等的，形式化的定义就是 P 与 P' 接收相同且合法的输入时会产生相同的输出；(2) P' 相比于 P 应该更加难以被攻击者分析。Gartner 的市场调研报告 [8] 显示，截止到 2017 年，有 18 所公司活跃于这个领域，其中有 8 所是创建于 2010 年以后的。许多商业产品中也使用了代码混淆技术，如 Skype 使用代码混淆防止攻击者逆向分析出它们的传输协议。由此可见，代码混淆是一种行之有效的软件保护技术。

随着符号执行与污点分析等软件形式化分析方法的发展，出现了许多有效的反混淆方法 [4, 14, 23, 27, 28]，传统代码混淆方法正面临着极大的挑战 [2]，因此提出针对这些分析方法的混淆技术是十分有必要的。而现有的针对符号执行的代码混淆方法 [2, 15, 18, 21, 25] 都存在严重缺陷。Banescu 等 [2] 提出在循环中插入可真可假型的不透明谓词以产生路径爆炸问题，以此来抵抗符号执行的攻击，Ollivier 等 [15] 提出的混淆方法也是基于路径爆炸问题引出的。他们使用 KLEE 验证所提出方法的有效性，但是 KLEE 并没有集成一些缓解爆炸问题的方案。Veritesting [1] 是一种缓解路径爆炸问题的方法，在遇到循环时，Veritesting 会进行循环展开，而后切换至静态符号执行，在执行完展开所得的代码后再切换至动态符号执行。我们的实验显示，集成了 Veritesting 的 Angr [19] 能处理他们提出的混淆方法。这些是第 1 类混淆方法，即对自动化反混淆方法没有明显抵抗性。Hui Xu 等 [25] 提出使用系统 API 构造混淆方法，现实世界中，符号执行工具不可能为每一个系统 API 建模，因此这种混淆方法是有效的，但是为系统 API 建模以处理这些混淆方法并不困难。这即是第 2 类混淆方法，即对自动化反混淆方法有抵抗性，但可以通过编写简单的扩展处理它们。Sharif 等 [18] 提出使用哈希算法加密分支条件，Zhi Wang 等 [21] 提出使用 3x+1 数学猜想引入路径爆炸问题。我们的实验显示，这些混淆方法是能有效抵抗自动化反混淆方法的，如果要使符号执行工具能够处理它们，只能通过提高约束处理器能力实现，我们认为这是十分困难的。它们是第 3 类混淆方法，即对自动化反混淆方法有抵抗性，而且很难通过扩展符号执行工具处理它们。然而，这些混淆方法引入的时间开销都比较高：(1) 哈希算法的开销较高是不言而喻的事实；(2) 当 3x+1 猜想的初始迭代数值较大时，需要数百次的迭代才能收敛，

同时也容易受到模式匹配攻击：(1) 识别加密算法可以使用代码相似度检测的方法，存在许多抗混淆的检测方法 [11, 20]，还存在针对加密算法识别的研究 [24]；(2) 3x+1 猜想的实现存在固定的模式。为了解决这些问题，我们深入研究了第 3 类混淆方法，提出了 3 种框架，使用这些框架，开发者可以构造出大量高强度的混淆方法。

许多反混淆方法都使用了污点分析 [16, 28]，还有一些反混淆方法 [14] 使用了程序切片，它们的作用都是识别出与输入或者某些数据相关的代码或指令。Yadegari 等 [26] 提出了比特级别的污点分析方法以改进基于污点分析的自动化反混淆方法，比特级别的污点分析能有效缓解代码混淆引入的过污染问题 [28]。在软件保护领域中，污点分析还被用来去除程序自校验 [3]。因此，提出针对污点分析的混淆方法是十分有必要的。我们提出了一种引入过污染问题的混淆方法，它也对程序切片存在抵抗作用。

本文的贡献在于：

1. 将已有反符号执行的混淆方法分为 3 类：(1) 无明显抵抗性；(2) 有抵抗性，但能通过编写简单的扩展处理；(3) 有抵抗性，并且无法通过编写简单的扩展处理。深入研究了第 3 类混淆方法，提出了 3 种框架，开发者可以使用这些框架构造出大量高强度的混淆方法，并且构造出的混淆方法时间开销更小

2. 提出了 1 种针对污点分析的混淆方法，它们可以产生过污染问题

本文的结构如下：

1. Sec.2是对现有的反自动化混淆方法的实证性研究，指出了部分混淆方法对符号执行的抵抗效果并不明显

2. Sec.3 介绍了自动化反混淆方法的背景，建立了威胁模型

3. Sec.4 介绍我们提出的 5 种混淆方法构造框架

4. Sec.5 评估了我们提出的混淆方法的有效性以及由此带来的时间开销与空间开销

```
void check(short c) {
  if ( c== 0xAABB)
    do_m();
}
```

(a) 原程序

```
int check(short c) {
  int v1, v2;
  for (v1=0; v1 < BYTE(c, 0); v1++);
  for (v2=0; v2 < BYTE(c, 1); v2++);
  if (v1 == 0xBB && v2 == 0xAA)
    do_m();
}
```

(b) 使用 FOR 混淆

图 1: FOR 混淆示例

2 实证研究

2.1 FOR 混淆

Ollivier 等 [15] 提出了一种基于路径爆炸的混淆方法，如图 1所示，用以抵抗符号执行的分析。路径爆炸是动态符号执行面临的挑战性问题，动态符号执行会为程序的每一条可能执行路径维护一个路径状态，当程序的可能执行路径过多时，则内存资源与计算资源都不足以支撑维护大量的路径状态。FOR 混淆通过在程序中插入 for 循环来实现数值转移，如图 1 (b) 所示，在第 1 个 for 循环执行完后，v1 的值等于 BYTE(c, 0)，其中，BYTE(c, n) 表示从 c 中第编号为 n 的字节的值，c 的低 8 位对应的字节编号为 0。每个 for 循环的可能执行次数是 255，那么，该程序中总共存在 255*255(65025) 条可能的可执行路径。很显然，当 for 混淆中存在的 for 循环越多时，则可能执行路径总数越大，混淆强度越高。如果使用最朴素的广度优先或者深度优先路径搜索方法，在我们的实验环境中，Angr 需要消耗半个小时的时间来覆盖所有可能的执行路径。Ollivier 等人基于这个事实，认为 FOR 混淆能有效抵抗符号执行工具的分析，他们并没有考虑到一些缓解路径爆炸问题的研究，如 Veritesting [1]。一般情况下，Veritesting 使用动态符号执行分析程序，当执行过程中遇见循环时，则尝试对循环进行展开，而后切换为静态符号执行，继续分析循环展开得到的代码，在执行完这些代码后，切换回动态符号执行。静态符号执行使用约束处理器中的 ITE 原语编码分支语义，本质是让约束处理器处理路径爆炸问题。图 2 (a) 是执行完循环展开后的程序，图 2 (b) 是 Veritesting 搜集的能触发 do _ m 函数执行的那条路径的路径约束，其中 ITE(C, a, b) 的语义是如果 C 为真，返回 a，反之，返回 b。

```
void check(short c) {
  int v1, v2;
  v1 = 0;
  if (0 < BYTE(c, 0))
    v1++;
  if (1 < BYTE(c, 1))
    v1++;
  if (2 < BYTE(c, 2))
    v1++;
  //.......
  v2 = 0;
  //.......
  if (v1 == 0xBB && v2 == 0xAA)
    do_m();
}
```

(a) 循环展开后的程序

```
v1 = 0 + ITE(0 < BYTE(c, 0), 1, 0)
  + ITE(1 < BYTE(c, 0), 1, 0)
  + ...
  + ITE(254 < BYTE(c, 0), 1, 0)
v2 = 0 + ITE(0 < BYTE(c, 1), 1, 0)
  + ITE(1 < BYTE(c, 1), 1, 0)
  + ...
  + ITE(254 < BYTE(c, 1), 1, 0)
v1 == 0xBB
v2 == 0xBB
```

(b) Veritesting 收集的路径约束

图 2: 使用 Veritesting 处理 FOR 混淆

Ollivier 等对 15 个 C 程序使用了 FOR 混淆（包含 1-4 个 for 循环），使用 KLEE 对它们进行攻击来证明混淆方法的有效性。这 16 个程序是数据集 1。这些程序都存在一条 Secret 分支，执行到这条分支时就会打印出”Successfully” 字符串，攻击成功的标志是符号执行工具寻找到能触发 Secret 分支的输入。KLEE 并没有实现 Veritesting 特性，因此我们使用 Angr 的 Veritesting 模块处理这些混淆程序，以验证 for 混淆对 Veritesting 的抵抗性。我们的实验结果如 1所示。Angr 成功攻击了数据集中的所有程序，且攻击消耗时间都在 5 分钟以内。Ollivier 等人的实验结果则显示，当 k=4 时，即 FOR 混淆中包含 4 个 for 循环时，KLEE 在 1 小时内没有攻破任何一个程序。Ollivier 等人只使用了朴素的 DFS、BFS、Random 这 3 种路径搜索方式，而没有考虑到其它缓解路径爆炸问题的策略。而我们的实验结果显示，Veritesting 可以处理 FOR 混淆。因此，我们将 FOR 混淆归入第 1 类，它对自动化反混淆方法没有明显的抵抗性。

程序	k=3	k=4	成功
file_0	82s	141s	√
file_3	85s	137s	√
file_5	93s	147s	√
file_7	87s	140s	√
file_10	123s	198s	√
file_12	170s	268s	√
file_15	80s	131s	√
file_18	122s	194s	√
file_19	81s	141s	√
file_22	83s	131s	√
file_28	123s	212s	√
file_35	80s	130s	√
file_39	82s	131s	√
file_45	83s	131s	√
file_47	80s	131s	√

表 1: Angr 处理 FOR 混淆

```
1 void check(int c) {
2   uint32_t j = c;
3   if (j == 7)
4     do_m();
5 }
```

(a) 原程序

```
1 void check(int c) {
2   uint32_t j = c;
3   int l1_ary[] = {1,2,3,4,5,6,7};
4   int l2_ary[] = {j,1,2,3,4,5,6,7};
5   int i = l2_ary[l1_ary[j % 7]];
6   if (i == 1 && j == 7)
7     do_m();
8 }
```

(b) 使用符号内存寻址混淆

图 3: 符号内存寻址混淆

2.2 符号内存寻址混淆

Hui Xu 等 [25] 提出了一种基于符号内存寻址的混淆方法，如图 3所示。符号内存寻址问题也是符号执行面临的挑战性问题，当程序访问的目标内存地址中包含符号值时，就产生了符号内存寻址问题。不同符号内存模型对符号内存寻址问题的处理方法不同。Angr 的默认策略是首先求出目标地址的取值范围，而后使用约束处理器的 ITE 原语编码每一种可能的访问情况。KLEE 也存在求解目标地址取值范围这一过程，这是为了检查是否存在内存访问越界错误。KLEE 是构建于 LLVM 工具链上的符号执行工具，LLVM IR 提供了诸如数组大小等信息，KLEE 在很多情况下可以直接获取目标地址的取值范围。图 3 (a) 是原程序，图 3 (b) 是混淆后的程序，l1 _ ary[j % 7] 的值等于 j % 7 + 1，l2 _ ary[j % 7 + 1] 的值等于 j % 7 + 1，因此 i == 1 等价于 j % 7 == 0，这是 j == 7 的必要条件。所以，i == 1 && j == 7 等价于 j == 7。因此混淆前后原程序语义等价，只有当 c 的值是 7 时才能

触发 do_m 函数的执行。Hui Xu 等的实验显示，Angr 无法处理这种混淆。

经过我们深入研究发现，当开启编译优化后，编译器会基于除数为常量的快速除法 [22] 实现求模运算，而不基于 div/idiv 指令。而 Angr 无法处理符号内存寻址混淆的原因就是 Claripy（基于 Z3-Solver 开发的约束求解器）无法在可接受时间范围内求出包含求模运算的目标地址的取值范围。Claripy 使用朴素的二分法实现求解最大值与最小值的接口，实际上，Z3-Solver 中集成有专门求解此类 MaxSAT 问题的 vZ 扩展 [5]，它能高效地处理 MaxSAT 问题。我们修改了 Claripy 的实现，使用 vZ 求解最大最小值。实验显示，我们修改后的 Angr 能有效处理符号内存寻址混淆，同时整个修改过程我们只编写了 30 行左右 Python 代码。因此，我们将这种混淆归入第 2 类，对符号执行工具有抵抗性，但可以通过编写简单的扩展来处理这种混淆。

2.3 哈希混淆

Sharif 等 [18] 提出了一种基于哈希加密相等/不等类条件分支的混淆方法，如图 4所示，图 (a) 是混淆前的原程序，图 (b) 是混淆后的程序，其中，HASH 代表广泛采用的哈希算法，如 MD5、SHA 系列等。由哈希算法的性质可知，c == 2022 等价于 HASH(c) == HASH(2022)，其中 HASH(2022) 的值是可以在编译时期计算的，换言之，在编译生成的可执行文件中，HASH(2022) 应该是一个已经确定的整数值。图 (b) 中还使用 2022 作为密钥加密了可执行文件中的基本块机器码，只有当分支触发时才使用 c 的值解密机器码。因此，如果攻击者不能寻找到正确的输入值 c，那么攻击者就无法解密被加密的机器码，也就无法进行进一步的分析。我们使用 MD5，SHA-1，SHA-256 混淆了 15 个 C 程序，以 1 小时作为超时时间，Angr 没有成功攻击任何一个程序。当 c 的类型是 int 时，通过暴力枚举法寻找到输入值 c 是可能的，但是当 c 的类型是 int64 时，暴力枚举就不再可行。以往的研究 [15, 21] 也认为约束处理器难以处理哈希算法。因此，我们将这种混淆分为第 3 类，对符号执行工具有明显抗性，并且难以通过编写简单的扩展处理。哈希混淆存在的问题是引入的哈希算法的开销过高，Z Wang 等 [21] 提出开销较低的线性混淆来代替哈希混淆。

```
void check(int c) {
  if (c == 2022)
    do_m();
}
```

(a) 原程序

```
void check(int c) {
  const int h = HASH(2022);
  int j = HASH(c);
  if (j == h) {
    Decrypt(BLOCK_CODE, c);
//Encrypted Basic Block Machine Code
  }
}
```

(b) 使用哈希混淆后

图 4: 哈希函数加密分支条件混淆

2.4 线性混淆

Z Wang 等 [21] 提出了一种基于 3x+1 未解数学猜想 [13] 的混淆方法，如图 5所示。3x+1 猜想的内容是：对于任意一个正整数 x，如果 x 是偶数，则把它除以 2，如果 x 是奇数，则乘以 3 再加 1，如此迭代若干次，最终 x 回到 1。图 5 (b) 中，对于 (c - m > 28) && (c + m < 32) 这一逻辑表达式，只有当 c = 30 且 m = 1 或 c 29, 30, 31 且 m = 0 时，它的值才会为真，而且 3x+1 猜想迭代过程中不会出现 0，m = 0 的情况不存在, 因此只有当 c = 30 且 m = 1 时才会触发这个分支，由 3x+1 猜想可知，m 最终一定会等于 1，因此混淆前后语义相等。

线性混淆也是基于路径爆炸问题的，但是我们的研究显示 Veritesting 无法处理线性混淆。Veritesting 中的一项重要属性是循环展开次数，当循环展开次数越多时，路径约束也就越复杂。我们使用 Angr 处理图 5 (b)，试图寻找到能触发 do_m 函数执行的输入。如图 6 所示，X 轴是循环展开次数，Y 轴是求解路径约束所需的时间。当循环展开次数小于 40 次时，约束处理器可以在极短时间内完成约束求解，当循环展开次数超过 40 次时，约束处理器求解所需的时间以指数形式增长。因此我们认为 Veritesting 无法有效处理线性混淆。

```
void check(int c) {
  if (c == 30)
    do_m();
}
```

(a) 原程序

```
void check(int32_t c) {
  int64_t m = c + 1000;
  while (m > 1) {
    if (m % 2)
      m = m * 3 + 1;
    else
      m /= 2;
    if ((c - m > 28) && (c + m < 32)
    )
      do_m();
  }
}
```

(b) 使用线性混淆后

图 5: 线性混淆

3 背景

3.1 自动化反混淆方法

随着符号执行方法的成熟，许多自动化且通用的反混淆方法被提出。Ming Jiang 等 [14] 提出了 LOOP，它以 TRACE 文件作为输入，输出检测出的不透明谓词；Yadegari 等 [27] 针对反混淆场景改进了符号执行方法，它能处理将直接跳转转换为间接跳转的混淆，而传统符号执行不能处理这类混淆代码。实验显示，改进后的符号执行能处理 VMProtect, Themida 等商用混淆工具产生的混淆代码；Yadegari 等 [28] 等提出了一种通用的反混淆方法，它与 LLVM 编译器的后端优化过程类似，它的输入是 TRACE 文件，而后对 TRACE 文件做代码简化，代码简化包含许多过程（类似于 LLVM 编译器的优化 PASS），最后将多个简化后的 TRACE 文件合并就可以重构出 CFG。为了重构完整的 CFG，必须要使用基于符号执行的路径探索技术。实验显示，这种通用的混淆方法能有效处理被 VMProtect，Themida，Code Virtualizer，EXECryptor 等商用混淆；Bardin 等 [4] 提出了后向动态符号执行，传统符号执行是前向的，用于解决路径可达性问题，后向符号执行是后向的，用于解决路径不可达问题，这恰好可以用来检测不透明谓词，而且后向符号执行还能处理代码自修改，栈顶返回地址修改等传统动态符号执行难以处理的混淆。还存在一些自动化但并不是通用的反混淆方法，它们只能处理某一类混淆，或者某一种混淆工具产生的混淆。Kan 等 [12] 提出的针

表 2: 反混淆方法涉及到的程序分析方法

反混淆方法	符号执行	污点分析	程序切片	TRACE
LOOP	√	√	√	√
Generic	√	√	×	√
BB-DSE	√	×	×	√
VM-Hunt	√	×	×	√
De-OLLVM	√	×	×	×

对 O-LLVM 混淆工具的反混淆方法；Dongpeng Xu 等 [23], Salwan 等 [16] 提出的针对代码虚拟化的反混淆方法。如表 2所示，列举了若干反混淆方法所使用到的程序分析方法。污点分析与程序切片的目的是提取出相关代码输入符号执行工具，这也能降低符号执行的开销。传统字节级污点分析工具在处理混淆时会面临非常严重的过污染问题，Yadegari 等 [26] 提出了比特级污点分析以解决这一问题。

3.2 威胁模型

在我们的威胁模型中，攻击者是使用自动化反混淆工具的人类攻击者，攻击目标就是被混淆的程序，并且攻击者只能获得可执行文件，而不能获得源代码。为了抵抗自动化反混淆工具，我们研究了已有的自动化反混淆方法，发现它们都基于符号执行、污点分析、程序切片这 3 种程序分析方法，因此，我们需要部署针对这些程序分析方法的特殊混淆。由于我们的攻击者中包含人类，我们的工作不能停留于抵抗工具的层面，还需要考虑人类分析者在攻击中所起到的作用。我们考虑两种情况：(1) 攻击者为 Angr 等分析框架编写扩展处理特殊混淆；(2) 攻击者编写反混淆脚本处理特殊混淆。对于 (1)，在 Sec.2中，我们通过实验将已有的一些工作分为 3 类，其中第 1，2 类都是不安全的，第 1 类是对自动化反混淆方法没有明显抵抗性，第 2 类是对自动化反混淆方法有抵抗性，但是可以通过编写简单的扩展处理它们。对于 (2)，我们调研了许多现实世界中的反混淆脚本，如针对 OLLVM 的控制流平坦化的反混淆脚本 [9]，它通过基本块前驱节点数目寻找分发块，分发块的特征是前驱节点数目最多。以及 CTF 比赛中的反混淆脚本 [10]，它通过指令序列作为特征识别出 Handle 作为后续分析的基础。

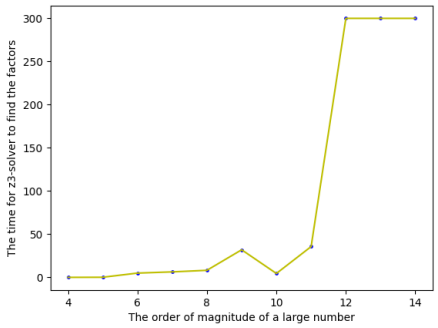

图 6: Z3-Solver 求解大整数分解难题所需时间

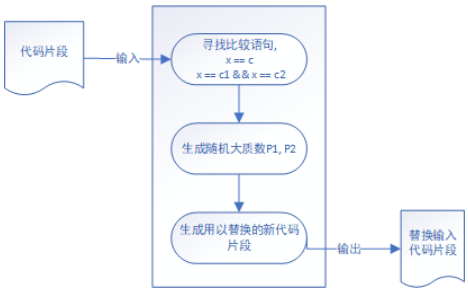

图 7: 基于大整数分解难题的构造框架

4 构造框架

4.1 基于大整数分解难题的构造框架

Sharif 等 [18] 提出使用哈希算法难以求逆这一公认的密码学难题来构造代码混淆方法，但是哈希算法的开销是高昂的。为了解决这个问题，我们转向另一个公认的密码学难题，即大整数分解难题。大整数分解的背景是：对于两个大质数 $p1, p2$，并且 $p1, p2$ 不过于接近，若只给出它们的乘积 m，将 m 分解为 $p1 \times p2$ 是非常困难的。如图 6所示是 Z3-Solver 求解大整数分解难题所需的时间，超时时间是 300s。当乘积 m 的数量级达到 10^{12} 时，约束求解器就难以在可接受的时间范围内返回结果。

如图 8所示，图 (a), (c) 是原程序，图 (b), (d) 是对应的混淆后的程序，P1 与 P2 是随机生成的大质数。在图 (d) 中，位于第 6 行的 if 语句是根据个位数判断 a + (P1 - 30) 的值等于 P1 或 P2。如图 7所示，是我们提出的构造框架，输入是代码片段，然后在输入中寻找可以混淆的特殊代码片段，随机生成若干质数对它进行混淆，最后输出混淆后的代码片段。

4.2 基于复合密码学原语的构造框架

这是构造 Sharif 等 [18] 提出的哈希混淆的框架。密码学原语指的是密码学中的基础组件，它们往往被组合使用作为一个安全的密码学算法中的一部分。良好的密码学算法往往考虑了方方面面的攻击手段，但是从目前已

```
void check(int a, int b) {
  if (a == 30 && b == 40)
    do_m();
}
```

(a) 原程序

```
void check(int a, int b) {
  if ((a + (P1 - 30)) * (b + (P2 -
     40)) == P1 * P2) {
    if (a + (P1 - 30) != 1 && b + (
     P2 - 30) != 1) {
      do_m();
    }
  }
}
```

(b) 混淆后

```
void check(int a, int b) {
  if (a == 30)
    do_m();
}
```

(c) 原程序

```
void check(int a, int b) {
  if (a + (P1 - 30) == 0 || (a + (P1
     - 30)) == P1*P2) {
    return;
  }
  if (P1*P2 % (a + (P1 - 30)) == 0)
     {
    if ((a + (P1 - 30)) % 10 != P2 %
     10) {
      do_m();
    }
  }
}
```

(d) 混淆后

图 8: 基于整数分解难题的混淆方法

知的反混淆手段来看，使用密码学原语构造单射函数时只需要考虑约束求解器的攻击即可。Sharif 等提出可以使用 CRC32、MD5、SHA256 等哈希算法，但是这些算法不只存在复杂约束这一类挑战性问题，还包括符号内存寻址，比如 CRC32 的查表实现，当 Feistel 网络中存在 if 分支时，还可能引入路径爆炸问题。当涉及到这两种问题时，无法确定我们构造的混淆的强度究竟来源于哪一类问题。为了避免这种情况，我们将要使用的密码学原语都不会引入这些挑战性问题。

- **Feistel 函数** 取 Feistel 网络结构的一轮作为 Feistel 函数。它的基本形式是 $L \leftarrow L \oplus G(R), t \leftarrow L,$
  $L \leftarrow R, R \leftarrow t, L \leftarrow L \oplus G(R)$，其中 L, R 是输入也是输出，函数形式是 $F(L,R) = (L \oplus G(R), R \oplus G(L \oplus G(R)))$，其中，G 可以是任意一个单射函数。可以将一个整形变量的高低两部分作为输入 L，R，将输出的 L 作为高位，R 作为低位，可以再组合形成一个整形变量，因此 $F(L,R)$ 可以看成是在整形变量上的单射函数。Feistel 函数的证明见附录 6.1节。

- **类 Feistel 函数** Feistel 函数中的异或运算可以重新定义为其它运算，只要满足某一特定性质，则导出的新函数仍然是单射的，我们将这些导出的单射函数称之为类 Feistel 函数。

- **仿射变换** 对于 $F(x) = ax + b \ mod \ m$，若 a, m 的最大公因数为 1，则它为仿射变换。在机器运算中，m 的值可以是 $2^8, 2^{16}, 2^{32}, 2^{64}$，对应着 char, short, int, int64 类型上的运算。

- **数据依赖的循环移位** 它的基本形式是 $a \leftarrow ROL(a \oplus b, b), b \leftarrow ROL(a \oplus b, a)$，其中 a, b 是输入也是输出，函数形式是 $F(L,R) = (ROL(L \oplus R, R), ROL(ROL(L \oplus R, R) \oplus b, ROL(L \oplus R, R)))$。其中，ROL 代表循环左移，也可以被替换为 ROR，即循环右移。

- **异或移位** $F(x) = x \oplus (x << c)$ 或 $F(x) = x \oplus (x >> c)$，其中，$<<$ 对应的是逻辑左移，也就是移位时低位补 0。

- **数据扩展** 将 n 字节的数据扩展至 m 字节的数据，其中 $m > n$。

复合使用上述密码学原语可以得到单射函数，因为单射函数的复合仍然是单射函数。单射函数的性质，有 $f(a) = f(b) => a = b, f(a) \neq f(b) => a \neq b$，

```
symvar = f1(symvar);
symvar = f2(symvar);
symvar = f3(symvar);
...
//这是可以在编译时计算出的常量
if (symvar == f1(f2(f3(CONST))))
...
```

图 9: 扩展哈希混淆

```
symvar += GE; //GE是随机的大整数
while (symvar > CONST) {
  if (symvar % M1 == 0) {
    if (symvar > CONST * N1)
      symvar /= N1;
    else if (symvar > CONST * N2)
      symvar /= N2;
    else if ...

    else
      symvar--;
  }
  else if ...

  else
    symvar--;
}
```

图 10: 扩展线性混淆

```
//x是int类型变量
SetByte(x, 0, table[GetByte(x, 0)]);
SetByte(x, 1, table[GetByte(x, 1)]);
SetByte(x, 2, table[GetByte(x, 2)]);
SetByte(x, 3, table[GetByte(x, 3)]);
//不透明谓词，永远不会成立
if (x * (x + 1) % 2 == 1)
```

图 11: 符号内存寻址片段混淆

```
//table1与table2是完全相同的数组
x1 = table1[x];
x2 = table2[x];
//z的值等于常量0
//污点分析，程序切片会认为x与z相关
//这引入了过污染问题
z = x2 - x1;
```

图 12: 过污染混淆

因此，我们可以使用单射函数替换高开销的哈希函数。基于这些，我们构造了一种特定的混淆方法，如图 9所示。我们提供一种高强度的单射函数作为示范：首先将 4 字节的输入扩展到 8 字节，这应用到了数据扩展原语，扩展过程中，4 字节的输入直接成为了扩展结果的低 32 位，因此该扩展过程一定是单射的；而后，进行 4 轮类 Feistel 函数加密，同时，每加密完一轮后都会进行仿射变换与数据依赖的循环移位加密。相比于 MD5、SHA256 等哈希算法，我们构造的混淆方法所引入的时间开销是非常低的，同时约束求解器也无法在可接受的时间范围内返回结果，这证明我们的方法是安全的。

4.3 扩展线性混淆

Z Wang 等 [21] 提出的线性混淆的核心部分是 3x+1 猜想，3x+1 猜想的内容是对于任意一个正整数，以某种方式不断迭代，迭代若干次后，一定会得到常数 1。我们提出了一种构造框架，它可以生成一个终止后一定能得到一个常数的循环，这与 3x+1 猜想非常相像。关于这点的证明见附录 6.2。

基于循环的混淆的构造框架与图 7很相近，它们都是输入代码片段，寻找可以混淆的特殊代码片段，随机生成若干数字对它进行混淆，输出混淆后的代码片段。如图 10所示，循环结束后，symvar 的值一定等于 CONST。值得注意的是 symvar += GE 这行代码，这是为了增加循环次数，基于循环的混淆方法的强度与循环次数高度相关。还有 M1，N1，N2 等都是随机生成的数字。使用该框架构造出的循环可以替代线性混淆中的 3x+1 猜想。

4.4 加强已有不透明谓词强度

利用符号内存寻址难题，我们可以构造出一种能显著影响约束求解器求解时间的代码片段。如图 11所示，代码中包含一个不透明谓词，$x \times (x+1)\%2$ 的值永远为 0，因此 if 分支，使用 Angr 检测出这种不透明谓词所需的时间是极短的 [25]。2-5 行代码是插入的符号内存寻址代码片段，GetByte(x, a) 表示取变量 x 的第 a 个字节值，SetByte(x, a, v) 表示设置变量 x 的第 a 个字节值为 v，table 数组中的元素是有序递增的。因此，SetByte(x, 0, table[GetByte(x, 0)]) 并不会改变 x 的值，但是会引入一次符号内存寻址难题。插入的代码片段对变量 x 的每个字节依次进行了去符号化，Angr 生成的路径约束形式是：$x0 = If(x0 = 0, 0, If(x0 = 1, 1, ...), x1 = If(x1 = 0, 0, If(x1 = 1, 1, ...), x2 = If(x2 = 0, 0, If(x2 = 1, 1, ...), x3 = If(x3 = 0, 0, If(x3 = 1, 1, ...), x = Contact(x0, x1, x2, x3), x * (x + 1)\ mod\ 2 \neq 0$. 生成的约束形式大约等价于让约束求解器暴力遍历每一种 x 的可能取值。实验显示，Angr 需要 1 分钟才能检测出被符号内存寻址加强过的不透明谓词，这极大增强了已有不透明谓词的强度。

4.5 产生过污染

污点分析或程序切片都是自动化反混淆方法中重要的组成部分，如 LOOP [14] 使用污点分析标记出与输入相关的代码来减少需要检测的谓词数，再使用程序切片提取出所有与谓词相关的代码进行符号执行，Yadegari

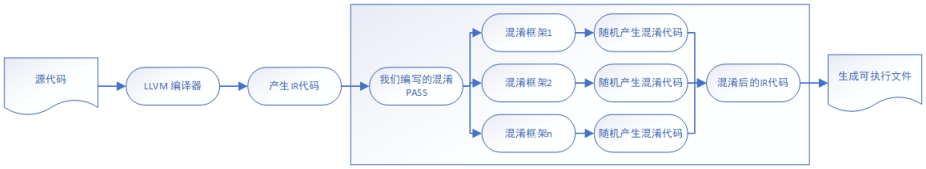

图 13: 混淆编译器流程

等提出的通用反混淆方法 [28] 使用污点分析提取出与程序输入相关的代码作为进一步反混淆的基础。在 LOOP 中，污点分析可以显著减少需要检测的谓词数，这使检测不透明谓词时间在可接受范围之内。在通用反混淆方法之中，Yadegari 等 [26] 提出了比特级别的污点分析方法，就是为了减少过污染问题对反混淆结果的影响，他们的实验显示，使用字节级别的污点分析方法处理混淆代码会产生严重的过污染问题，导致反混淆结果极不理想。已有的研究 [6] 都关注于引入少污染问题，而在混淆中，引入过污染问题对反混淆方法才有更显著的作用。我们提出了一种低开销的引入过污点问题的混淆方法，如图 12所示。table1 与 table2 是元素相同的数组，因此变量 z 的值是常数 0，但是如果在变量 z 使用后向污点分析，那么 x 会被标记为污染，在变量 x 使用前向污点分析，那么 z 会被标记为污染。这种混淆方法只引入了两次内存访问指令与一条减法运算指令，开销很低，因此可以在程序中大范围部署，引入大量的过污染问题。

5 实验评估

我们基于 LLVM 工具链实现了一个混淆编译器，它可以生成带混淆的可执行文件，它的工作流程如图 13所示。我们基于 LLVM 工具链编写了一个 Pass，它输入 LLVM IR，输出混淆后的 LLVM IR，最终由 LLVM 编译器后端生成可执行文件。

5.1 空间开销

我们编写了一个 C 程序，它包含标准 AES 实现，以及若干对输入的检验。我们使用混淆编译器编译了该程序，以 $\frac{Size_{Original}-Size_{Obfuscated}}{Size_{Original}}$ 作为空间开销百分比。实验结果如图 14所示。

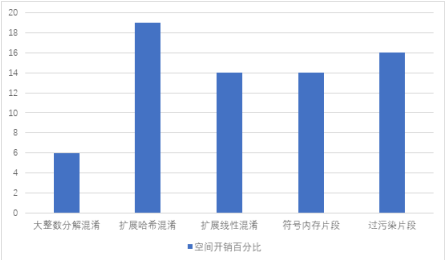

图 14: 空间开销

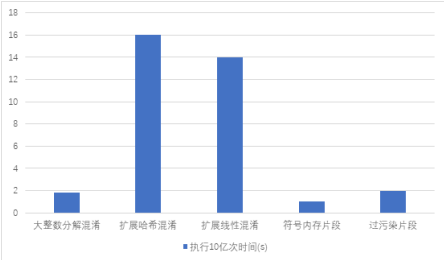

图 15: 引入代码片段时间开销

5.2 时间开销

如果将混淆代码片段插入循环中，那么带来的时间开销是很大的。由于我们设计的混淆编译器对于混淆位置的选取是随机的，因此使用混淆编译器产生可执行文件而后比较运行时间开销是不合理的。我们考虑应用混淆方法后在原程序中增加的代码片段，而后统计这些代码片段的执行时间即可明确混淆方法的时间开销。实验结果如图 15所示。

5.3 对自动化分析工具的抗性

Ollivier 等 [15] 提供了一个数据集，包含 15 个 C 程序，这些 C 程序都包含一个分支，如果符号执行工具能够找到能触发这个分支的输入就视为符号执行工具攻击成功。我们使用混淆编译器依次使用大整数分解混淆，扩展哈希混淆，扩展线性混淆产生了 45 个混淆后的可执行文件，而后使用 Angr 验证混淆强度。Angr 在 5 小时内没有成功攻击 45 个混淆程序中的任何一个。

我们研究了符号内存代码片段对加强现有不透明谓词抵抗符号执行的效果。表 3显示插入符号内存寻址代码片段后检测不透明谓词所需的时间，但是，随着不透明谓词中涉及的符号变量数的增加，抵抗强度的增加是线性的。而后，我们尝试在不透明谓词涉及到的符号变量之间建立约束关系，如表 4所示，如果我们在符号变量之间引入约束关系能显著提升强度。

我们使用二进制分析平台 Triton [17] 验证过污染混淆对污点分析的抵抗性。我们使用 Triton 分析如图 12所示的程序，在第 7 行代码处开始对 z 进行前向污点分析，Triton 错误地将变量 x 标记为污点源，成功引入了过污染问题。

表 3: 符号变量增加

不透明谓词	符号变量数	原检测时间	混淆后检测时间
x * (x + 1) % 2 != 0	1	<1s	63s
7 * y * y - 1 == x * x	2	<1s	221s
2x - y - z != (x ^y) + 2 * (x | (~y)) + 2 + (~z - x)	3	<1s	392s

表 4: 符号变量之间存在约束关系

不透明谓词	符号变量数	符号变量约束	混淆后检测时间
7 * y * y - 1 == x * x	2	x = x + y	221s
2x - y - z != (x ^y) + 2 * (x | (~y)) + 2 + (~z - x)	3	z = x + y + z	>3h (未输出结果)

5.4 对模式匹配攻击的抗性

由于我们提出的不仅仅是一种混淆方法，而是一种混淆构造框架，软件开发者可以基于我们的框架构造出更多的混淆方法，这足以抵抗模式匹配攻击。例如，符号内存寻址代码片段以及过污染混淆都会引入具有连续元素的数组，因此，内存中存在连续递增的元素可以作为模式特征。但是结合我们在前文提出的密码学原语可以解决这个问题，如图 16所示，对于数组 table 而言，它的索引到元素值可以认为是单射函数 f 的逆映射。

```
//f是使用密码学原语构造的单射函数
//x是1字节长度变量
for (int i = 0; i < 256; i++) {
  table[f(i)] = i;
}

x = table[x];
x = f(x);
```

图 16: 引入不连续数组减少模式特征

参考文献

[1] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1083–1094, 2014.

[2] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200, 2016.

[3] S. Banescu, S. Valenzuela, M. Guggenmos, M. Ahmadvand, and A. Pretschner. Dynamic analysis versus obfuscated self-checking. In *Annual Computer Security Applications Conference*, pages 182–193, 2021.

[4] S. Bardin, R. David, and J.-Y. Marion. Backward-bounded dse: targeting infeasibility questions on obfuscated codes. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 633–651. IEEE, 2017.

[5] N. Bjørner and A.-D. Phan. νz-maximal satisfaction with z3. *Scss*, 30:1–9, 2014.

[6] L. Cavallaro, P. Saxena, and R. Sekar. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. *Secure Systems Lab at Stony Brook University, Tech. Rep*, pages 1–18, 2007.

[7] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Citeseer, 1997.

[8] M. B. Dionisio Zumerle, Manjunath BhatDionisio Zumerle. Market guide for application shielding. Technical report, Gartner, 2017.

[9] F. Gabriel. Deobfuscation: recovering an ollvm-protected program, 2014.

[10] hxp. Confidence ctf 2015: Reversing 400 ”deobfuscate me” writeup, 2015.

[11] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu. Value-based program characterization and its application to software plagiarism detection. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 756–765, 2011.

[12] Z. Kan, H. Wang, L. Wu, Y. Guo, and D. X. Luo. Automated deobfuscation of android native binary code. *arXiv preprint arXiv:1907.06828*, 2019.

[13] J. C. Lagarias. The 3 x+ 1 problem and its generalizations. *The American Mathematical Monthly*, 92(1):3–23, 1985.

[14] J. Ming, D. Xu, L. Wang, and D. Wu. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 757–768, 2015.

[15] M. Ollivier, S. Bardin, R. Bonichon, and J.-Y. Marion. How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections). In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 177–189, 2019.

[16] J. Salwan, S. Bardin, and M.-L. Potet. Symbolic deobfuscation: From virtualized code back to the original. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 372–392. Springer, 2018.

[17] F. Saudel and J. Salwan. Triton: Concolic execution framework. In *Symposium sur la sécurité des technologies de l' information et des communications (SSTIC)*, 2015.

[18] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*. Citeseer, 2008.

[19] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016*

IEEE Symposium on Security and Privacy (SP), pages 138–157. IEEE, 2016.

[20] Z. Tian, Q. Zheng, T. Liu, and M. Fan. Dkisb: Dynamic key instruction sequence birthmark for software plagiarism detection. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 619–627. IEEE, 2013.

[21] Z. Wang, J. Ming, C. Jia, and D. Gao. Linear obfuscation to combat symbolic execution. In *European Symposium on Research in Computer Security*, pages 210–226. Springer, 2011.

[22] H. S. Warren. *Hacker's delight*. Pearson Education, 2013.

[23] D. Xu, J. Ming, Y. Fu, and D. Wu. Vmhunt: A verifiable approach to partially-virtualized binary code simplification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 442–458, 2018.

[24] D. Xu, J. Ming, and D. Wu. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 921–937. IEEE, 2017.

[25] H. Xu, Y. Zhou, Y. Kang, F. Tu, and M. Lyu. Manufacturing resilient bi-opaque predicates against symbolic execution. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 666–677. IEEE, 2018.

[26] B. Yadegari and S. Debray. Bit-level taint analysis. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 255–264. IEEE, 2014.

[27] B. Yadegari and S. Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 732–744, 2015.

[28] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*, pages 674–691. IEEE, 2015.

6 附录

6.1 类 Feistel 函数单射性证明

使用反证法证明，不妨假设 Feistel 函数不是单射的，那么存在 $I_0 = (L_0, R_0), I_1 = (L_1, R_1), I_0 \neq I_1, f(I_0) = f(I_1), i.e., L_0 \oplus G(R_0) = L_1 \oplus G(R_1), R_0 \oplus G(L_0 \oplus G(R_0)) = R_1 \oplus G(L_1 \oplus G(R_1))$。而后我们进行分类讨论。

- $L_0 \neq L_1, R_0 \neq R_1$ 有 $R_0 \oplus G(L_0 \oplus G(R_0)) = R_1 \oplus G(L_0 \oplus G(R_0))$，显然，由异或运算性质可推出 $R_0 = R_1$，与已知条件矛盾。
- $L_0 = L_1, R_0 \neq R_1$ 有 $G(R_0) = G(R_1)$，则有 $R_0 = R_1$，与已知条件矛盾。
- $L_0 \neq L_1, R_0 = R_1$ 有 $L_0 \oplus G(R_0) = L_1 \oplus G(R_0)$，由异或运算性质可推出 $L_0 = L_1$，与已知条件矛盾。

综上所述，Feistel 函数是单射的。

6.2 基于循环的混淆的收敛证明

由循环体内对 symvar 的赋值可知，symvar 的值是不断减小的，因此，该循环一定会终止。不妨假设循环终止时，$symvar < CONST$。那么就有，$symvar \div N1 < CONST$ 或 $symvar \div N2 < CONST$ 或 $symvar - 1 < CONST$，它们分别与 $symvar > CONST \times N1$, $symvar > CONST \times N2$, $symvar > CONST$ 矛盾。因此，循环终止时，必然有 $symvar = CONST$。