加密实验

对 OLLVM 和 Tigress 两种混淆工具的混淆方法进行整理，结果如表 1-1 所示。

表 1-1　OLLVM 和 Tigress 混淆方法

	OLLVM	Tigress
混淆方法	(1) bogus control flow (2) instructions substitution (3) control flow flattening	(1) Virtualize (2) Jit　(将函数转换为运行时生成的代码，不确定) (3) JitDynamic　(将函数转换为动态形式，不确定) (4) Flatten (5) Split (6) Merge (7) Add Opaque (8) Encode Literals (9) Encode Data (10) Enc. Arithmetic (11) Encode External (12) Encode Branches (13) AntiAliasAnalysis (14) AntiTaintAnalysis (15) Opaque Predicate (16) Generate Entropy (收集随机性变量，需要结合其他变换如(18) ×) (17) Inplicit Flow (18) Random Function (19) Clean Up (20) Info　(打印内部信息×) (21) Copy (22) Leak Information　(不确定) (23) Ident　　(未实现×) (24) Randomize Args (25) Top Level　　(执行 tigress 的必要选项 ×) (26) Self Modify (在执行时自行修改，表现为部分代码被删除或被其他指令替换)　(代码替换，指令重构) (27) Checksum　(添加检查器，检测程序完整性×) (28) Inline (29) Optimize (程序内优化，某些变换如(28)后使用×) (30) Measure、Software Metrics (复杂度及插件×) (31) Plugins (支持插件×)

针对传统混淆技术，选取 OLLVM 和 Tigress 两种混淆工具并对它们的功能进行分类，具体如表 1-2 所示。

表 1-2　OLLVM 和 Tigress 功能分类

传统混淆技术			混淆器		文献（代码混淆技术综述. 20220603）
类	子类	方法	OLLVM	Tigress	
布局混淆		删除		√(19)	[18]

		改名			[19-20]
控制流混淆	计算变换	不透明谓词混淆	√(1)	√(15)	[21-32]
		冗余代码插入	√(1)	√(17)	[33-35]
		库调用混淆		√(11)	[36-37]
		将可约控制流图转化为不可约控制流图			
		代码并行化			
		扩展循环条件		√(7)	[38]
		语义等价变换	√(2)	√(10)	[39-45]
	聚合变换	方法内嵌与外联		√(28)	[46]
		方法重叠与分割		√(5)(6)	[47]
		方法复制与代理		√(21)	[48]
		循环变换			
		分支函数		√(12)	[49-59]
		控制流扁平化	√(3)	√(4)	[60-67]
	顺序变换			√(18)	[68-70]
数据流混淆	数据编码混淆			√(9)	[71-74]
	数据重构混淆	变量重构			
		数组重构			
		类重构			[75-79]
	数据随机化			√(24)	[80-82]
	将静态数据转换为过程			√(8)	[83]
预防混淆	内在预防变换				
	针对性预防变换			√(13) (14)	[84-87]
新型混淆技术		虚拟化		√(1)	

1 OLLVM

使用 OLLVM 的控制流扁平化功能进行 C 程序混淆，它会将整个程序的控制流进行扁平化。在 OLLVM 的 github 网址（https://github.com/obfuscator-llvm/obfuscator/wiki/Control-Flow-Flattening）上，控制流扁平化这一功能的实现引用了论文“Obfuscating C++ programs via control flow flattening, Annales Univ. Sci. Budapest., Sect. Comp. 30 (2009) 3-19.”。此文献的扁平化过程如下：先将函数体划分为基本块，再将不同嵌套层级的基本块放在一起，这些基本块被封装在选择结构（即 switch 语句）中，最后通过一个表示程序状态的控制变量来确保正确的控制流程。然而，OLLVM 与上述文献仍有差别，OLLVM 完全扁平化控制流而上述文献只扁平化某个函数的控制流。

具体地，OLLVM 使用 switch 语句扁平化控制流并利用 switch 变量维持基本块的执行流程。使用 ida 查看 RSA 算法 OLLVM 混淆后的二进制程序 RSA_obf_O，使用 F5 查看 main 函数部分伪代码，发现其控制流程是使用 switch 语句实现的。

```
switch ( v14 )
{
  case -1200763609:
    v3 = strlen(s);
    v4 = -1688327938;
    if ( v17 < v3 )
      v4 = -1885068116;
    v14 = v4;
    break;
  case -1066655961:
    v11 = strlen(s);
    v12 = 251268422;
    if ( v17 < v11 )
      v12 = 1453826455;
    v14 = v12;
    break;
  case -897665432:
    ++v17;
    v14 = 461801203;
    break;
  case -214071935:
    fprintf(stderr, "Error in encryption!\n");
    v26 = 1;
    v14 = -1678658684;
    break;
  case 251268422:
    printf("\n");
    free(ptr);
```

1.1 RSA

实现 RSA 算法的程序为 RSA.c。在 main 函数中定义字符型输入数据 message，得到加密后的整型加密数据 encrypted 及解密后的字符型解密数据 decrypted。但是，message 与 decrypted 是以整型 printf 输出的。此外，primes.txt 为质数文件，从此 txt 文件中随机选取的质数可以组成公钥和私钥。具体如图 1-1 所示。

```
struct public_key_class pub[1];
struct private_key_class priv[1];
char *PRIME_SOURCE_FILE="primes.txt";
rsa_gen_keys(pub, priv, PRIME_SOURCE_FILE);

printf("Private Key:\n Modulus: %lld\n Exponent: %lld\n", (long long)priv->modulus, (long long) priv->exponent);
printf("Public Key:\n Modulus: %lld\n Exponent: %lld\n", (long long)pub->modulus, (long long) pub->exponent);

char message[] = "123abc";
int i;

printf("Original:\n");
for(i=0; i < strlen(message); i++){
  printf("%lld\n", (long long)message[i]);
}

long long *encrypted = rsa_encrypt(message, sizeof(message), pub);
if (!encrypted){
  fprintf(stderr, "Error in encryption!\n");
  return 1;
}
printf("Encrypted:\n");
for(i=0; i < strlen(message); i++){
  printf("%lld\n", (long long)encrypted[i]);
}

char *decrypted = rsa_decrypt(encrypted, 8*sizeof(message), priv);
if (!decrypted){
  fprintf(stderr, "Error in decryption!\n");
  return 1;
}
printf("Decrypted:\n");
for(i=0; i < strlen(message); i++){
  printf("%lld\n", (long long)decrypted[i]);
}
```

图 1-1 RSA.c 部分代码

再使用 gcc 编译生成二进制程序 RSA，执行命令为'./RSA'。

接着，使用 clang 对 RSA.c 程序进行控制流扁平化，得到混淆后的程序 RSA_obf_O。具体如图 1-2 所示。

```
[root@FF 1_RSA]# clang RSA.c -o RSA_obf_O -mllvm -fla
```

图 1-2 OLLVM 混淆 RSA.c 命令

1.1.1 正解性

分别执行原程序 RSA 和混淆程序 RSA_obf_O，以验证混淆程序的正解性。具体如图 1-3 及表 2-1 所示。

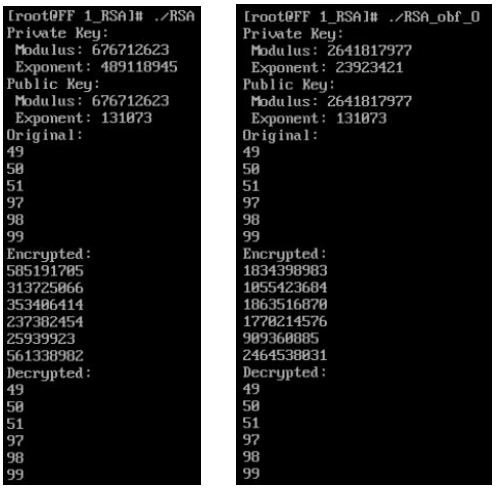

图 1-3 RSA 及 RSA_obf_O 运行结果

表 2-1 RSA 及 RSA_obf_O 的正解性

	RSA	RSA_obf_O
正解性	√	√

从图 1-3 可以看出，OLLVM 混淆后的 RSA_obf_O 能正确进行加解密。

1.1.2 强度

（1）程序长度

基于 IDA Pro 的 IDAEye 插件，进行 IDA 编程以计算每个函数中操作符及操作数的个数，结果如表 2-2 所示。

表 2-2 使用自编写代码测量 RSA 及 RSA_obf_O 的程序长度

函数	RSA	RSA_obf_O
rsa_encrypt	56+97=153	129+220=349
rsa_decrypt	98+173=271	222+381=603
main	137+235=372	282+489=771
gcd	18+27=45	43+69=112
extEuclid	48+84=132	71+122=193
modmult	91+141=232	229+379=608
rsa_modExp	44+71=115	136+229=365
rsa_gen_keys	234+400=634	447+770=1217
程序长度	1954	4218

（2）控制流循环复杂度

使用 angr 测量 RSA 和 RSA_obf_O 的控制流循环复杂度，结果如下。

表 2-3 使用 angr 测量 RSA 及 RSA_obf_O 的控制流循环复杂度

函数	RSA	RSA_obf_O
rsa_encrypt	4	11
rsa_decrypt	7	18
main	6	18
gcd	2	4
ExtEuclid	2	4
modmult	12	22
rsa_modExp	6	12

rsa_gen_keys	13	35
控制流循环复杂度	52	124

1.1.3 执行代价

（1）程序大小

表 2-4 RSA 及 RSA_obf_O 程序大小

	RSA	RSA_obf_O
程序大小（KB）	14	18

RSA 的程序大小为 14KB，RSA_obf_O 的程序大小为 18KB。

（2）运行时间

表 2-5 RSA 及 RSA_obf_O 运行时间

运行时间 ms	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10	avg
RSA	10	20	30	20	30	10	10	20	10	30	19
RSA_obf_O	180	120	70	40	70	40	140	40	90	40	83

RSA 的运行时间为 19 毫秒，RSA_obf_O 的运行时间为 83 毫秒。

1.1.4 隐蔽性

使用 ida 插件 Diaphora 测试程序的隐蔽性（相似性）。

表 2-6 RSA 及 RSA_obf_O 相似性

函数	RSA 与 RSA_obf_O 相似性
rsa_encrypt	18%
rsa_decrypt	20%
main	27%
gcd	50%
extEuclid	82%
modmult	23%
rsa_modExp	26%
rsa_gen_keys	19%
Avg similarity	33.125%

1.2 MD5

实现 MD5 算法的程序为 MD5.c。字符型输入数据 msg 需要用户通过参数输入，加密后的数据 result 为整型。具体如图 2-1 所示。

```
char *msg;
size_t len;
int i;
uint8_t result[16];

if (argc < 2) {
    printf("usage: %s 'string'\n", argv[0]);
    return 1;
}
msg = argv[1];

len = strlen(msg);

// benchmark
for (i = 0; i < 1000000; i++) {
    md5((uint8_t*)msg, len, result);
}

// display result
for (i = 0; i < 16; i++)
    printf("%2.2x", result[i]);
```

图 2-1 MD5.c 部分代码

再使用 gcc 编译生成二进制程序 MD5，直接用'./MD5 xxx'执行程序即可。

接着，使用 clang 对 MD5.c 程序进行控制流扁平化，得到混淆后的程序 MD5_obf_O。具体如图 2-2 所示。

图 2-2 OLLVM 混淆 MD5.c 命令

1.2.1 正解性

分别执行原程序 MD5 和混淆程序 MD5_obf_O，以验证混淆程序的正解性。具体如图 2-3 及表 3-1 所示。

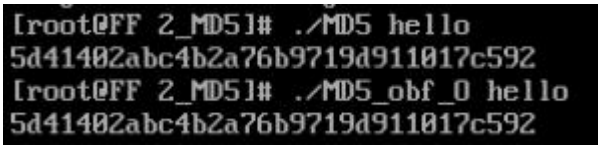

图 2-3 MD5 和 MD5_obf_O 运行结果

表 3-1 MD5 及 MD5_obf_O 的正解性

	MD5	MD5_obf_O
正解性	√	√

从图 2-3 可以看出，OLLVM 混淆后的 MD5_obf_O 能正确进行加密。

1.2.2 强度

（1）程序长度

基于 IDA Pro 的 IDAEye 插件，进行 IDA 编程以计算每个函数中操作符及操作数的个数，结果如表 3-2 所示。

表 3-2 使用自编写代码测量 MD5 及 MD5_obf_O 的程序长度

函数	MD5	MD5_obf_O
md5	216+396=612	440+778=1218
main	51+84=135	147+248=395
程序长度	747	1613

（2）控制流循环复杂度

使用 angr 测量 MD5 及 MD5_obf_O 的控制流循环复杂度表，结果如下所示。

表 3-3 使用 angr 测量 MD5 及 MD5_obf_O 的控制流循环复杂度

函数	MD5	MD5_obf_O
md5	9	30
main	4	13
控制流循环复杂度	13	43

1.2.3 执行代价

（1）程序大小

表 3-4 MD5 及 MD5_obf_O 程序大小

	MD5	MD5_obf_O
程序大小（KB）	13	13

MD5 的程序大小为 13KB，MD5_obf_O 的程序大小为 13KB。

（2）运行时间

表 3-5 MD5 及 MD5_obf_O 运行时间

运行时间 ms	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10	avg
MD5	1240	1240	1240	1260	1260	1280	1270	1240	1280	1270	1258

MD5_obf_O	21700	22640	21760	22280	22140	21930	22340	22110	21410	21640	21995

其中，待加密数据为‘helloworld!’。MD5 的运行时间为 1258 毫秒，MD5_obf_O 的运行时间为 21995 毫秒。

1.2.4 隐蔽性

使用 ida 插件 Diaphora 测试程序的隐蔽性（相似性）。

表 3-6 MD5 及 MD5_obf_O 相似性

函数	MD5 与 MD5_obf_O 相似性
md5	24%
main	16%
Avg similarity	20%

1.3 AES

实现 AES 算法的程序为 AES.c。在 main 函数中定义十六进制的整型输入数据 pt 和初始密钥 key，通过 aesEncrypt 函数加密后得到整型加密数据 ct，再使用 aesDecrypt 函数进行解密以得到整型解密数据 plain。具体如图 3-1 所示。

```
const uint8_t key[16] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
const uint8_t pt[16] = {0x32, 0x43, 0xf6, 0xa8, 0x88, 0x5a, 0x30, 0x8d, 0x31, 0x31, 0x98, 0xa2, 0xe0, 0x37, 0x07, 0x34};
uint8_t ct[16] = {0};
uint8_t plain[16] = {0};

aesEncrypt(key, 16, pt, ct, 16);
printHex(pt, 16, "plain data: ");
printHex(ct, 16, "after encryption:");

aesDecrypt(key, 16, ct, plain, 16);
printHex(plain, 16, "after decryption:");
```

图 3-1 AES.c 部分代码

再使用 gcc 编译生成二进制程序 AES，此程序的执行命令为’./AES’。

接着，使用 clang 对 AES.c 程序进行控制流扁平化，得到混淆后的程序 AES_obf_O。具体如图 3-2 所示。

```
[root@FF 3_AES]# clang AES.c -o AES_obf_O -mllvm -fla
```

图 3-2 OLLVM 混淆 AES.c 命令

1.3.1 正解性

分别执行原程序 AES 和混淆程序 AES_obf_O，以验证混淆程序的正解性。具体如图 3-3 及表 4-1 所示。

```
[root@FF 3_AES]# ./AES
plain data:
data[16]: 32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34
after encryption:
data[16]: 39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
after decryption:
data[16]: 32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34
[root@FF 3_AES]# ./AES_obf_O
plain data:
data[16]: 32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34
after encryption:
data[16]: 39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32
after decryption:
data[16]: 32 43 F6 A8 88 5A 30 8D 31 31 98 A2 E0 37 07 34
```

图 3-3 AES 和 AES_obf_O 运行结果

表 4-1 AES 及 AES_obf_O 的正解性

	AES	AES_obf_O
正解性	√	√

从图 3-3 可以看出，OLLVM 混淆后的 AES_obf_O 能正确进行加解密。

1.3.2 强度

（1）程序长度

基于 IDA Pro 的 IDAEye 插件，进行 IDA 编程以计算每个函数中操作符及操作数的个数，结果如表 4-2 所示。

表 4-2 使用自编写代码测量 AES 及 AES_obf_O 的程序长度

函数	AES	AES_obf_O
main	71+132=203	59+107=166
loadStateArray	29+48=77	93+157=250
storeStateArray	29+48=77	93+157=250
keyExpansion	194+356=550	371+660=1031
addRoundKey	65+108=173	112+195=307
subBytes	35+54=89	95+161=256
invSubBytes	35+54=89	95+161=256
shiftRows	115+192=307	153+276=429
invShiftRows	115+192=307	153+276=429
GMul	29+50=79	107+185=292
mixColumns	121+200=321	218+379=597
invMixColumns	121+200=321	218+379=597
aesEncrypt	113+194=307	305+527=832
aesDecrypt	115+198=313	307+531=838
printHex	32+52=84	69+118=187
printState	51+82=133	73+126=199
程序长度	3430	6916

（2）控制流循环复杂度

使用 angr 测量 AES 及 AES_obf_O 的控制流循环复杂度表，结果如下所示。

表 4-3 使用 angr 测量 AES 及 AES_obf_O 的控制流循环复杂度

函数	AES	AES_obf_O
main	1	1
loadStateArray	3	9
storeStateArray	3	9
keyExpansion	8	26
addRoundKey	3	9
subBytes	3	9
invSubBytes	3	9
shiftRows	2	9
invShiftRows	2	9
GMul	4	9
mixColumns	5	17
invMixColumns	5	17
aesEncrypt	8	19
aesDecrypt	8	19
printHex	2	5
printState	2	5
控制流循环复杂度	62	181

1.3.3 执行代价

（1）程序大小

表 4-4 AES 及 AES_obf_O 程序大小

	AES	AES_obf_O
程序大小（KB）	18	22

AES 的程序大小为 18KB，AES_obf_O 的程序大小为 22KB。

（2）运行时间

表 4-5 AES 及 AES_obf_O 运行时间

运行时间 ms	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10	avg
AES	0	0	0	0	0	10	10	10	10	10	5
AES_obf_O	0	0	0	0	0	10	10	10	10	10	5

AES 的运行时间为 5 毫秒，AES_obf_O 运行时间为 5 毫秒。

1.3.4 隐蔽性

使用 ida 插件 Diaphora 测试程序的隐蔽性（相似性）。

表 4-6 AES 及 AES_obf_O 相似性

函数	AES 及 AES_obf_O 相似性
main	20%
loadStateArray	30%
storeStateArray	31%
keyExpansion	13%
addRoundKey	20%
subBytes	20%
invSubBytes	20%
shiftRows	18%
invShiftRows	21%
GMul	30%
mixColumns	21%
invMixColumns	17%
aesEncrypt	20%
aesDecrypt	23%
printHex	36%
printState	13%
Avg similarity	22.0625%

2 Tigress

使用 Tigress 的控制流扁平化功能进行 C 程序混淆，它只会扁平化指定函数的控制流。Tigress 官网（https://tigress.wtf/flatten.html）上的控制流扁平化的实现引用了以下几个论文。

- Flattening first appears in Chenxi Wang's thesis.
- The llvm obfuscator supports a flattening transformation.
- Laszlo and Kiss, Obfuscating C++ Programs via Control Flow Flattening
- Udupa et al, Deobfuscation: Reverse engineering obfuscated code.

其中，第一个是由 Wang Chenxi 首先提出控制流平化的论文，它先将控制结构分解为 if-then-go 结构，

再修改 goto 语句以保证 goto 跳转正确。具体在 C 语言中实现时，使用 switch 语句来扁平化控制流，再通过 switch 变量确定下一个基本块的位置。第二个是 OLLVM 的控制流扁平化链接。第三个是论文“Obfuscating C++ programs via control flow flattening, Annales Univ. Sci. Budapest., Sect. Comp. 30 (2009) 3-19.”。第四个是反混淆论文“Deobfuscation: Reverse engineering obfuscated code”。

具体地，Tigress 使用 switch 语句扁平化控制流并利用 switch 变量维持基本块的执行流程。使用查看 RSA 算法 Tigress 混淆后的 C 程序 RSA_obf_T.c，发现其控制流程是使用 switch 语句实现的。

```
while (1) {
  switch (_1_main_next) {
  case 18: ;
  return (1);
  break;
  case 25:
  PRIME_SOURCE_FILE = (char *)"primes.txt";
  rsa_gen_keys(pub, priv, PRIME_SOURCE_FILE);
  printf((char const   */* __restrict   */)"Private Key:\n Modulus: %lld\n Exponent: %lld\n",
         priv[0].modulus, priv[0].exponent);
  printf((char const   */* __restrict  */)"Public Key:\n Modulus: %lld\n Exponent: %lld\n",
         pub[0].modulus, pub[0].exponent);
  message[0] = (char )'1';
  message[1] = (char )'2';
  message[2] = (char )'3';
  message[3] = (char )'a';
  message[4] = (char )'b';
  message[5] = (char )'c';
  message[6] = (char )'\000';
  printf((char const   */* __restrict  */)"Original:\n");
  i___0 = 0;
  {
  _1_main_next = 5UL;
  }
  break;
  case 4: ;
  if (! encrypted) {
    {
    _1_main_next = 8UL;
    }
  } else {
    {
    _1_main_next = 9UL;
    }
  }
  break;
```

2.1 RSA

使用 tigress 对 RSA.c 程序进行控制流扁平化，得到混淆后的程序 RSA_obf_T.c。具体如图 4-1 所示。

```
[root@FF 1_RSA]# tigress --Environment=x86_64:Linux:Gcc:7.3.1 --Transform=Flatten --Functions=rsa_en
crypt,rsa_decrypt,main,gcd,ExtEuclid,modmult,rsa_modExp,rsa_gen_keys --out=RSA_obf_T.c RSA.c
```

图 4-1 Tigress 混淆 RSA.c 命令

为保证混淆程序的一致性，使用 gcc 编译 RSA_obf_T.c 以得到二进制程序 RSA_obf_T。

2.1.1 正解性

分别执行原程序 RSA 和混淆程序 RSA_obf_T，以验证混淆程序的正解性。具体如图 4-2 及表 5-1 所示。

```
[root@FF 1_RSA]# ./RSA
Private Key:
 Modulus: 676712623
 Exponent: 489118945
Public Key:
 Modulus: 676712623
 Exponent: 131073
Original:
49
50
51
97
98
99
Encrypted:
585191705
313725066
353406414
237382454
25939923
561338982
Decrypted:
49
50
51
97
98
99
```

```
[root@FF 1_RSA]# ./RSA_obf_T
Private Key:
 Modulus: 3782802259
 Exponent: 1204182129
Public Key:
 Modulus: 3782802259
 Exponent: 131073
Original:
49
50
51
97
98
99
Encrypted:
2568465432
1710148893
1823202544
2816003700
2388308972
1146086132
Decrypted:
49
50
51
97
98
99
```

图 4-2 RSA 及 RSA_obf_T 运行结果

表 5-1 RSA 及 RSA_obf_T 的正解性

	RSA	RSA_obf_T
正解性	√	√

从图 4-2 可以看出，Tigress 混淆后的 RSA_obf_T 能正确进行加解密。

2.1.2 强度

Tigress 提供了测量程序长度和控制流循环复杂度的功能，使用以下代码对 RSA.c 及控制流扁平化后的 RSA.c 进行测量。具体如下图所示。

```
[root@FF 1_RSA]# tigress --Environment=x86_64:Linux:Gcc:7.3.1 --Transform=SoftwareMetrics --Function
s=rsa_encrypt,rsa_decrypt,main,gcd,ExtEuclid,modmult,rsa_modExp,rsa_gen_keys --SoftwareMetricsFileNa
me=rsa.txt --SoftwareMetricsKind=raw,halstead,mccabe --Transform=Flatten --Functions=rsa_encrypt,rsa
_decrypt,main,gcd,ExtEuclid,modmult,rsa_modExp,rsa_gen_keys --Transform=SoftwareMetrics --Functions=
rsa_encrypt,rsa_decrypt,main,gcd,ExtEuclid,modmult,rsa_modExp,rsa_gen_keys --SoftwareMetricsFileName
=rsa_obf_t.txt --SoftwareMetricsKind=raw,halstead,mccabe RSA.c --out=obf_t.c_
```

其中，此程序输出 rsa.txt、rsa_obf_t.txt 及 obf_t.c，rsa.txt 为 RSA.c 的强度指标文件，rsa_obf_t.txt 为控制流扁平化后的 RSA.c 的强度指标文件，obf_t.c 是控制流扁平化后的 RSA.c 与强度指标相关的 C 文件。具体指标如表 5-2 所示。

（1）程序长度

基于 IDA Pro 的 IDAEye 插件，进行 IDA 编程以计算每个函数中操作符及操作数的个数，结果如表 5-2 所示。

表 5-2 使用自编写代码测量 RSA 及 RSA_obf_T 的程序长度

函数	RSA	RSA_obf_T
rsa_encrypt	56+97=153	87+146=233
rsa_decrypt	98+173=271	165+279=444
main	137+235=372	209+347=556
gcd	18+27=45	38+56=94
extEuclid	48+84=132	71+117=188
modmult	91+141=232	140+215=355
rsa_modExp	44+71=115	85+133=218
rsa_gen_keys	234+400=634	315+540=855
程序长度	1954	2943

（2）控制流循环复杂度

表 5-3 使用 angr 测量 RSA 及 RSA_obf_T 的控制流循环复杂度

函数	RSA	RSA_obf_T
rsa_encrypt	4	16
rsa_decrypt	7	30
main	6	29
gcd	2	6
ExtEuclid	2	7
modmult	12	31
rsa_modExp	6	20
rsa_gen_keys	13	46
控制流循环复杂度	52	185

2.1.3 执行代价

（1）程序大小

表 5-4 RSA 及 RSA_obf_T 程序大小

	RSA	RSA_obf_T
程序大小（KB）	14	18

RSA 的程序大小为 14KB，RSA_obf_T 的程序大小为 18KB。

（2）运行时间

表 5-5 RSA 及 RSA_obf_T 运行时间

运行时间 ms	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10	avg
RSA	10	20	30	20	30	10	10	20	10	30	19
RSA_obf_T	30	40	10	30	30	20	20	20	50	30	28

RSA 的运行时间为 19 毫秒，RSA_obf_T 的运行时间为 28 毫秒。

2.2.2 强度

Tigress 提供了测量程序长度和控制流循环复杂度的功能，使用以下代码对 MD5.c 及控制流扁平化后的 MD5.c 进行测量。具体如下图所示。

```
[root@FF 2_MD5]# tigress --Environment=x86_64:Linux:Gcc:7.3.1 --Transform=SoftwareMetrics --Function
s=md5,main --SoftwareMetricsFileName=md5.txt --SoftwareMetricsKind=raw,halstead,mccabe --Transform=F
latten --Functions=md5,main --Transform=SoftwareMetrics --Functions=md5,main --SoftwareMetricsFileNa
me=md5_obf_t.txt --SoftwareMetricsKind=raw,halstead,mccabe MD5.c --out=obf_t.c_
```

其中，此程序输出 md5.txt、md5_obf_t.txt 及 obf_t.c，md5.txt 为 MD5.c 的强度指标文件，md5_obf_t.txt 为控制流扁平化后的 MD5.c 的强度指标文件，obf_t.c 是控制流扁平化后的 MD5.c 与强度指标相关的 C 文件。具体指标如下所示。

（1）程序长度

基于 IDA Pro 的 IDAEye 插件，进行 IDA 编程以计算每个函数中操作符及操作数的个数，结果如表 6-2 所示。

表 6-2 使用自编写代码测量 MD5 及 MD5_obf_T 的程序长度

函数	MD5	MD5_obf_T
md5	216+396=612	281+499=780
main	51+84=135	94+153=247
程序长度	747	1027

（2）控制流循环复杂度

表 6-3 使用 angr 测量 MD5 及 MD5_obf_T 的控制流循环复杂度

函数	MD5	MD5_obf_T
md5	9	34
main	4	17
控制流循环复杂度	13	51

2.1.4 隐蔽性

使用 ida 插件 Diaphora 测试程序的隐蔽性（相似性）。

表 5-6 RSA 及 RSA_obf_T 相似性

函数	RSA 与 RSA_obf_T 相似性
rsa_encrypt	47%
rsa_decrypt	56%
main	54%
gcd	52%
extEuclid	64%
modmult	74%
rsa_modExp	61%
rsa_gen_keys	54%
Avg similarity	57.75%

2.2 MD5

使用 tigress 对 MD5.c 程序进行控制流扁平化，得到混淆后的程序 MD5_obf_T.c。具体如图 5-1 所示。

```
[root@FF 2_MD5]# tigress --Environment=x86_64:Linux:Gcc:7.3.1 --Transform=Flatten --Functions=main,m
d5 --out=MD5_obf_T.c MD5.c
```

图 5-1 Tigress 混淆 MD5.c 命令

为保证混淆程序的一致性，使用 gcc 编译 MD5_obf_T.c 以得到二进制程序 MD5_obf_T。

2.2.1 正解性

分别执行原程序 MD5 和混淆程序 MD5_obf_T，以验证混淆程序的正解性。具体如图 5-2 及表 6-1 所示。

```
[root@FF 2_MD5]# ./MD5 hello
5d41402abc4b2a76b9719d911017c592
[root@FF 2_MD5]# ./MD5_obf_T hello
5d41402abc4b2a76b9719d911017c592
```

图 5-2 MD5 及 MD5_obf_T 运行结果

表 6-1 MD5 及 MD5_obf_T 的正解性

	MD5	MD5_obf_T
正解性	√	√

从图 5-2 可以看出，Tigress 混淆后的 MD5_obf_T 能正确进行加解密。

2.2.3 执行代价

（1）程序大小

表 6-4 MD5 及 MD5_obf_T 程序大小

	MD5	MD5_obf_T
程序大小（KB）	13	13

MD5 的程序大小为 13KB， MD5_obf_T 的程序大小为 13KB。

（2）运行时间

表 6-5 MD5 及 MD5_obf_T 运行时间

运行时间 ms	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10	avg
MD5	1240	1240	1240	1260	1260	1280	1270	1240	1280	1270	1258
MD5_obf_T	2640	2650	2600	2600	2690	2720	2670	2660	2650	2650	2653

MD5 的运行时间为 1258 毫秒，MD5_obf_T 的运行时间为 2653 毫秒。

2.2.4 隐蔽性

表 6-6 MD5 及 MD5_obf_T 相似性

函数	MD5 与 MD5_obf_T 相似性
md5	52%
main	45%
Avg similarity	48.5%

2.3 AES

使用 tigress 对 AES.c 程序进行控制流扁平化，得到混淆后的程序 AES_obf_T.c。具体如图 6-1 所示。

```
[root@FF 3_AES]# tigress --Environment=x86_64:Linux:Gcc:7.3.1 --Transform=Flatten --Functions=main,l
oadStateArray,storeStateArray,keyExpansion,addRoundKey,subBytes,invSubBytes,shiftRows,invShiftRows,G
Mul,mixColumns,invMixColumns,aesEncrypt,aesDecrypt,printHex,printState,main --out=AES_obf_T.c AES.c
```

图 6-1 Tigress 混淆 AES.c 命令

为保证混淆程序的一致性，使用 gcc 编译 AES_obf_T.c 以得到二进制程序 AES_obf_T。

2.3.1 正解性

分别执行原程序 AES 和混淆程序 AES_obf_T，以验证混淆程序的正解性。具体如图 6-2 及表 7-1 所示。

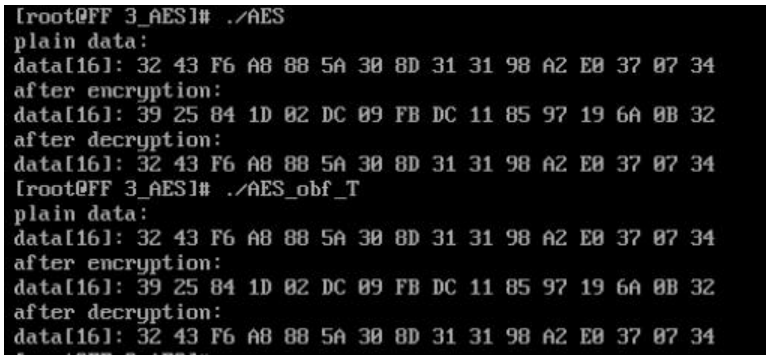

图 6-2 AES 及 AES_obf_T 运行结果

表 7-1 AES 及 AES_obf_T 的正解性

	AES	AES_obf_T
正解性	√	√

从图 6-2 可以看出，Tigress 混淆后的 AES_obf_T 能正确进行加解密。

2.3.3 强度

Tigress 提供了测量程序长度和控制流循环复杂度的功能，使用以下代码对 AES.c 及控制流扁平化后的 AES.c 进行测量。具体如下图所示。

```
[root@FF 3_AES]# tigress --Environment=x86_64:Linux:Gcc:7.3.1 --Transform=SoftwareMetrics --Function
s=main,loadStateArray,storeStateArray,keyExpansion,addRoundKey,subBytes,invSubBytes,shiftRows,invShi
ftRows,GMul,mixColumns,invMixColumns,aesEncrypt,aesDecrypt,printHex,printState --SoftwareMetricsFile
Name=aes.txt --SoftwareMetricsKind=raw,halstead,mccabe --Transform=Flatten --Functions=main,loadStat
eArray,storeStateArray,keyExpansion,addRoundKey,subBytes,invSubBytes,shiftRows,invShiftRows,GMul,mix
Columns,invMixColumns,aesEncrypt,aesDecrypt,printHex,printState --Transform=SoftwareMetrics --Functi
ons=main,loadStateArray,storeStateArray,keyExpansion,addRoundKey,subBytes,invSubBytes,shiftRows,invS
hiftRows,GMul,mixColumns,invMixColumns,aesEncrypt,aesDecrypt,printHex,printState --SoftwareMetricsFi
leName=aes_obf_t.txt --SoftwareMetricsKind=raw,halstead,mccabe AES.c --out=obf_t.c_
```

其中，此程序输出 aes.txt、aes_obf_t.txt 及 obf_t.c，aes.txt 为 AES.c 的强度指标文件，aes_obf_t.txt 为控制流扁平化后的 AES.c 的强度指标文件，obf_t.c 是控制流扁平化后的 AES.c 与强度指标相关的 C 文件。具体指标如下所示。

（1）程序长度

基于 IDA Pro 的 IDAEye 插件，进行 IDA 编程以计算每个函数中操作符及操作数的个数，结果如表 7-2 所示。

表 7-2 使用自编写代码测量 AES 及 AES_obf_T 的程序长度

函数	AES	AES_obf_T
main	71+132=203	120+213=333
loadStateArray	29+48=77	54+85=139
storeStateArray	29+48=77	54+85=139
keyExpansion	194+356=550	256+451=707
addRoundKey	65+108=173	89+145=234
subBytes	35+54=89	59+91=150
invSubBytes	35+54=89	58+89=147
shiftRows	115+192=307	151+248=399
invShiftRows	115+192=307	151+248=399
GMul	29+50=79	60+97=157
mixColumns	121+200=321	159+263=422
invMixColumns	121+200=321	159+262=421
aesEncrypt	113+194=307	234+386=620

aesDecrypt	115+198=313	236+390=626
printHex	32+52=84	51+82=133
printState	51+82=133	69+110=179
程序长度	3430	5205

（2）控制流循环复杂度

表 7-3 使用 angr 测量 AES 及 AES_obf_T 的控制流循环复杂度

函数	AES	AES_obf_T
main	1	12
loadStateArray	3	11
storeStateArray	3	19
keyExpansion	8	32
addRoundKey	3	11
subBytes	3	11
invSubBytes	3	11
shiftRows	2	14
invShiftRows	2	14
GMul	4	14
mixColumns	5	19
invMixColumns	5	19
aesEncrypt	8	47
aesDecrypt	8	47
printHex	2	7
printState	2	8
控制流循环复杂度	62	296

2.3.2 执行代价

（1）程序大小

表 7-4 AES 及 AES_obf_T 程序大小

	AES	AES_obf_T
程序大小（KB）	18	26

AES 的程序大小为 18KB，AES_obf_T 的程序大小为 26KB。

（2）运行时间

表 7-5 AES 及 AES_obf_T 运行时间

运行时间 ms	run1	run2	run3	run4	run5	run6	run7	run8	run9	run10	avg
AES	0	0	0	0	0	10	10	10	10	10	5
AES_obf_T	0	0	0	0	0	10	10	10	10	10	5

AES 的运行时间为 5 毫秒，AES_obf_T 运行时间为 5 毫秒。

2.3.4 隐蔽性

使用 ida 插件 Diaphora 测试程序的隐蔽性（相似性）。

表 7-6 AES 及 AES_obf_T 相似性

函数	AES 及 AES_obf_T 相似性
main	34%
loadStateArray	44%

storeStateArray	46%
keyExpansion	76%
addRoundKey	79%
subBytes	66%
invSubBytes	67%
shiftRows	63%
invShiftRows	62%
GMul	47%
mixColumns	44%
invMixColumns	44%
aesEncrypt	44%
aesDecrypt	44%
printHex	54%
printState	80%
Avg similarity	55.875%