CTF中的代码变形技术和难度分析

王晨男（大龙猫） 鲁大师

2021 SDC分会场-公开课

本讲内容

1，简单案例，理论方法初体验

2，方案设计和难度分析的理论工具

3，攻与守的常见套路介绍和分析

4，一种实践成功的套路

5，总结

简单案例

方案场景：Base64 编码/解码，已知输出求输入。

保护方案：采用自定义编码符号 + 代码膨胀。

$$f((a_{i,1}, a_{i,2}, a_{i,3})) = (b_{i,1}, b_{i,2}, b_{i,3}, b_{i,4})$$

简单案例

方案场景：Base64 编码/解码，已知输出求输入。

保护方案：采用自定义编码符号 + 代码膨胀。

攻击手法1：黑盒方法。我们发现改变输入的一个位，对输出的影响极为局部化，所以对输出的每4位（如果是Base64解码，那么用3位）可以蛮力穷举出局部输入。整体输入是局部输入的简单拼接，复杂度相对于输入数据长度是线性的。

攻击手法1.X（真实实战案例）：无脑操作。对输出的每1位，蛮力穷举输入，就行了。

如果是编码，求输入每次遇到多解，下一步匹配会自动丢弃错误解；如果是解码，每次蛮力确定1到2位输入

简单案例

方案场景：Base64 编码/解码，已知输出求输入。

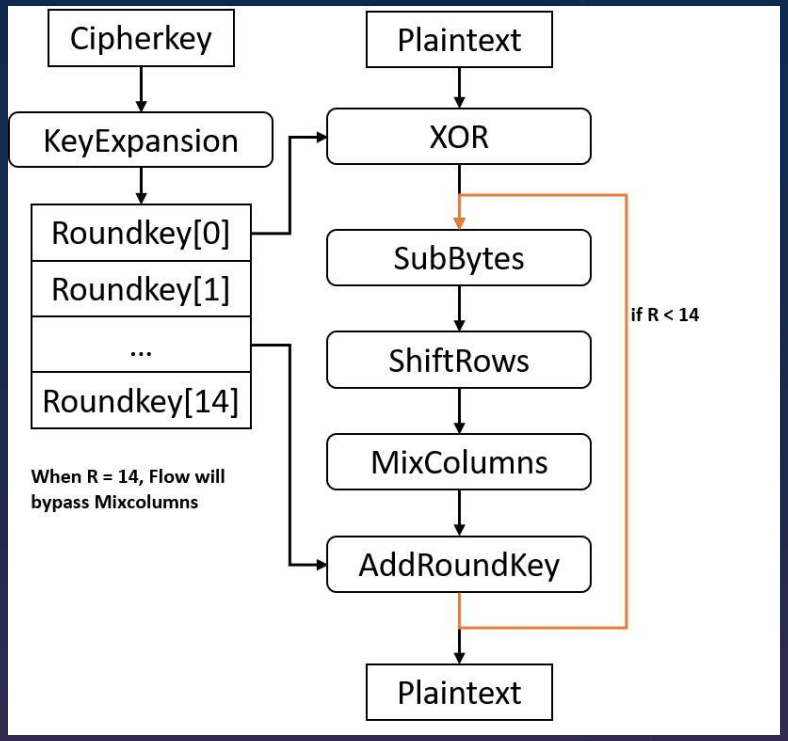
保护方案：采用自定义编码符号 + 代码膨胀。

攻击手法2：人肉蛮力。汇编层面，逐层逆推。

假设场景为编码，代码膨胀是汇编指令的等价替换，那么构成单输入单输出双射，即回推过程中一个a的6bits对应一个已知的b，只需机械操作，无需人工分析。

假设场景为解码，找到构成双射的局部代码块，或者局部处理一下小规模多解，问题也不难。

简单案例



https://xilinx.github.io/Vitis_Libraries/security/2020.1/_images/original_flow.png

方案场景：标准AES，已知输出求输入。KCTF方案2.

保护方案：

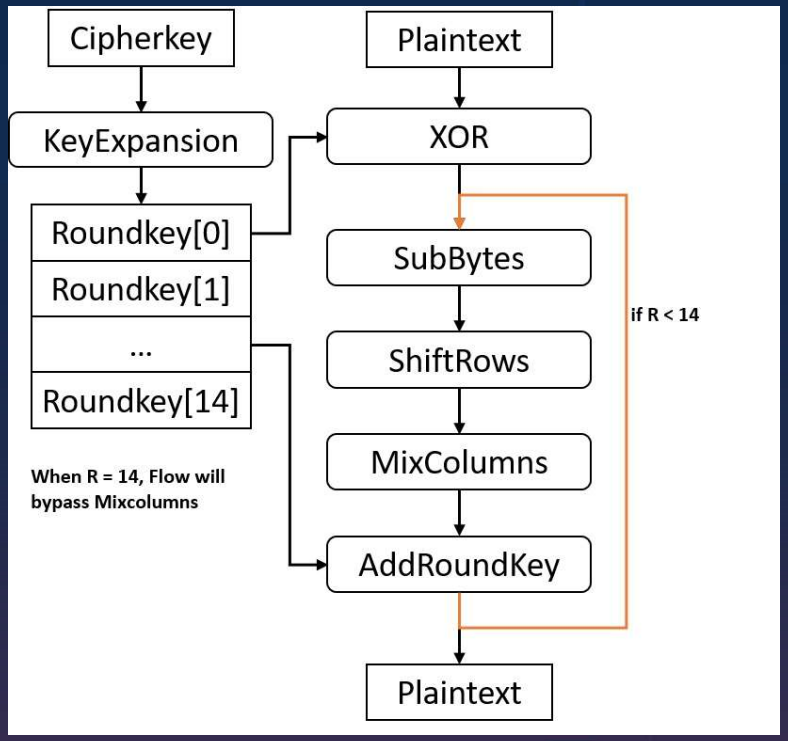
简易类Themida虚拟机。

为Sub, ShiftRow, MixColumn, Xor分配handler.

handler内部采用代码膨胀，

由于RoundKey固定，AddRoundKey的密钥逐字节Xor转化为查表。

简单案例



https://xilinx.github.io/Vitis_Libraries/security/2020.1/_images/original_flow.png

方案场景：标准AES，已知输出求输入。KCTF方案2.

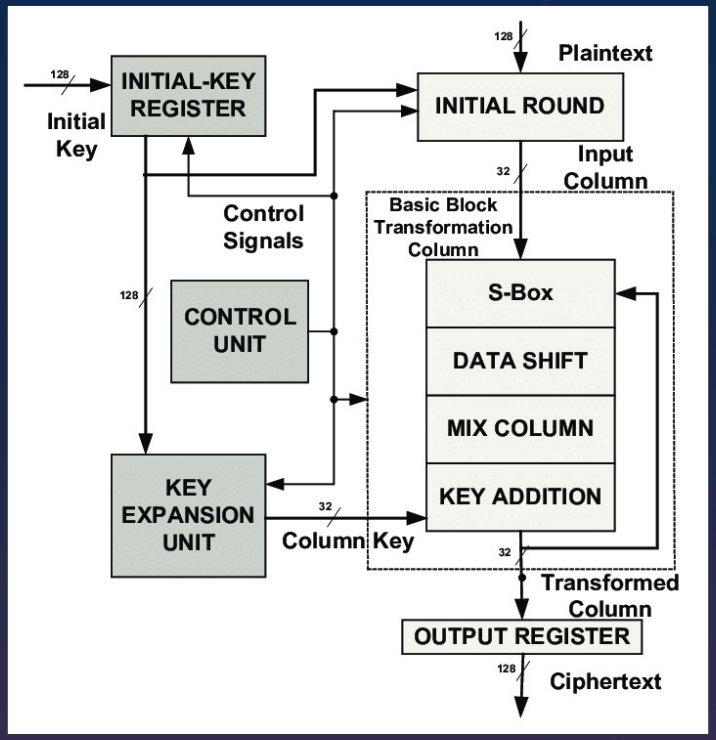
攻击方法：

根据示例答案的数据流得到执行顺序，重建算法Sub Shift, MixColumn, AddRoundKey, 循环的分层。

逐层逆推。

AddRoundKey逆映射简单，直接求得逆映射，无需关心密钥是什么。

简单案例



https://www.researchgate.net/profile/Nicolas-Sklavos/publication/228656196/figure/fig2/AS:670002449494028@1536752440629/Advanced-Encryption-Standard-AES-Scheme-Architecture.png

方案场景：标准AES，已知输出求输入。KCTF方案2.

保护方案：制成硬件。

如果不抠开硬件看电路，我没有什么办法。

这个方案除了题目不过审没有什么问题。

电线上监听信号/能耗特征，不属于本讲范围。

这个无聊案例，有其理论意义！

理论工具

1，映射建模

2，值域分析

3，香农信息论

4，归约证明

5，复杂度评估

映射建模

CTF题目，就是以输入为变量，经过一系列计算，得出答案正确或答案错误这个二值结果，结果集合记作 {true, false}.

通用形式是：

答案正确与否 = f(username, key)

常见的表现形式是：

答案正确与否 = equal(f(username, key), 常数)

答案正确与否 = equal(f(username, key), g(username))

映射建模

映射可以拆解为映射的复合，最终得到如下形式：

f : A -> B　　f = equal fn-1 … f3 f2 f1 T

T　　f1　　f2　　f3　　fn-1　　equal
A -> A1 -> A2 -> A3 -> … -> An -> B

B 中的 true 的原象，和原象的原象，一直到最初的正确输入，组成的链，最终有唯一解。

Tips:

当无关的算法构成分支，视为分段映射

f: D -> R

D = D1 ∪ D2, f1: D1 -> R1, f2: D2 -> R2, R = R1 ∪ R2

映射建模

1，对于 f(x) = y 的双射，求逆映射的难度?
蛮力穷举复杂度
数学公式求逆

2，对于 f(a, b) = y 的非双射 g，组织成双射。
当 g 被识别，则转化为问题1
g 的识别可以有难度

本质上是 f_i({x_i}) = y_i 的方程组。

3，对于 f(a, b) = y 的非双射 g，目标值有唯一原象
下一节值域分析

4，对于分界明显的映射序列，难度等同于最难的一步（千层套娃除外）

值域分析

对于已建模的映射，当其输入满足某有意义的约束的，通过分析其值域的范围，获得可利用的约束。

约束的获取，和收紧，可以利用有解且有唯一解的性质。

定义域需要是用户输入可覆盖的，不然难以建立约束。

例：

f: D1 ∪ D2 -> R1 ∪ R2,　f1: D1 -> R1, f2: D2 -> R2, I = R1 ∩ R2

当 I 为空时，目标结果或目标结果的原象/目标结果所在子集的原象位于 R1.

模型中去掉 D2 及后续过程。

当 I 不为空时，如果目标结果或目标结果的原象位于 I 内，则此题多解。（隐含前提是D1,D2可以被用户输入覆盖）

模型中去掉 I 的原象及后续过程。

值域分析

以下是一些建立约束可以着手的点：

设计层面的有效输入和无效输入对值域的影响（真实翻车事件，目标结果所在子集的原象）
局部映射来自于用户输入的部分和来自于状态机、干扰项的输入
设计层面多态函数的控制位的组合（抗值域分析是题目设计难点）

从映射层面砍掉大量代码和大幅缩小有效输入范围

任何算法都要落地为分步执行的指令流，利用计算过程中的约束可以做很多事情
小到查bug，大到Hash碰撞

香农信息论

$$H(X) = -\sum_{x \in \mathbb{N}} p(x) \log p(x)$$

CTF crackme中涉及的一个实体x，对应集合A中的一个元素。

我们想知道：x对应的是A中的哪一个元素？

没有数据，我们只能盲猜。

我们得到了和x相关的数据d，d对于我们回答上一个问题提供了帮助，比盲猜更准确——d具有信息。

香农信息论

$$H(X) = -\sum_{x \in \mathcal{N}} p(x) \log p(x)$$

信息是可以量化的，但我们不用真的去计算，而是利用一些性质。

防守方试图擦除信息，攻击方试图获取信息。

分析的关键在于构建意义的集合和定义数据描述，并且清晰的列出。

香农信息论

A = {handler对应的运算}，对于虚拟机

A(x) = {是不是运算的边界?是/否}，对于代码膨胀、门电路集成

A1(x) = {真实块，虚假块}， A2 = 基本块执行序列的集合，对于CFF（控制流平坦化）场景

只要攻击方想知道的问题q的答案，所有可能的答案就构成集合A_q.

攻击方通过数据去判断实体的归属，即x_q是A_q中的哪一个元素。从判断x_q更可能属于A_q的某个子集开始，逐步缩小范围。

防守方要做的是让这个判断尽可能等可能的分配于集合A的元素之中。此时信息熵最大，信息最少。

变形的本质是擦除信息，创造不可区分性。（工程上常说的歧义性）

香农信息论

让各种形态彼此长得像，是通过数据去看的。

代码在如下方面提供数据。（包括不限于）

由于代码的可执行性:

指令使用，局部输入输出

由于代码的结构性：

跳转关系，流程图，AST

由于名设计：

模型约束在代码块和控制流中的体现

香农信息论

任何一个问题可以归结为，从代码到数据，通过分析数据来做实体归类的问题。
这是擦除/获取信息的主战场。

清晰定义问题的三要素（d, x, A），根据问题“如何由 d 来判断 x 是 A 中的哪个元素”来设计和攻击方案，可以事半功倍，更加严密。

归约证明

如果我们攻破了A，那么我们就攻破了B.

B往往是某个数学难题，或者名算法。

常见变体：

假设我们解决了A，基于其算法，我们把解决B的复杂度降低了。

易错：

算法B的实现用到了A != 解决A，就需要解决B != 解决B就可以解决A

错误的分析往往导致：黑盒旁路攻击

正确的分析往往导致：此CTF题不成立

复杂度评估

复杂度是一个学科。

CTF中常见的复杂度评估在于蛮力穷举（指数）和有效的功能单元组合数量（Catalan数）。

值得注意的是，如果归约证明没有做，那么评估出的复杂度实际上只是“自己能想到的攻击手段的复杂度”而已。

出题方要同时避免复杂度过低和过高。

对于确信破解复杂度过高的地方，可以认为这不是预设的突破口。

某个旁路复杂度过低，这可能就是出题人设计不周，果断拿flag——这事情见得多了。

攻与守的常见套路和分析

我们利用理论工具，

从最简单的由VS编译的debug版AES程序进化出我们常见的所有套路和一些可能的改进方案。

场景：

aes(f(username, flag), aes_key) = g(username)

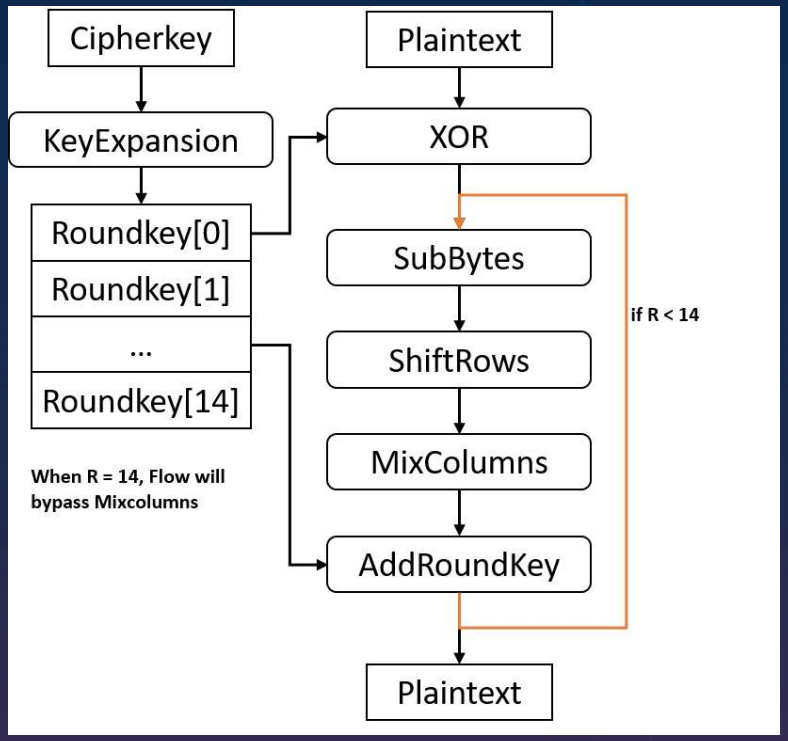
aes_key是写死的

如果 aes_key = h(username) 那么，其中一处映射等价于上式（映射建模）

如果 aes_key = h(username, key) 那么，由于数学难题，作者自己也解不出方案2（归约证明）

关键步骤的映射：aes(input) = 定值

赤裸状态



aes = ∏aes_round

aes_round = AddRoundKey mss

mss = MixColumns ShiftRows SubBytes

每一步都是双射

AddRoundKey 只要是双射就行，形式不固定

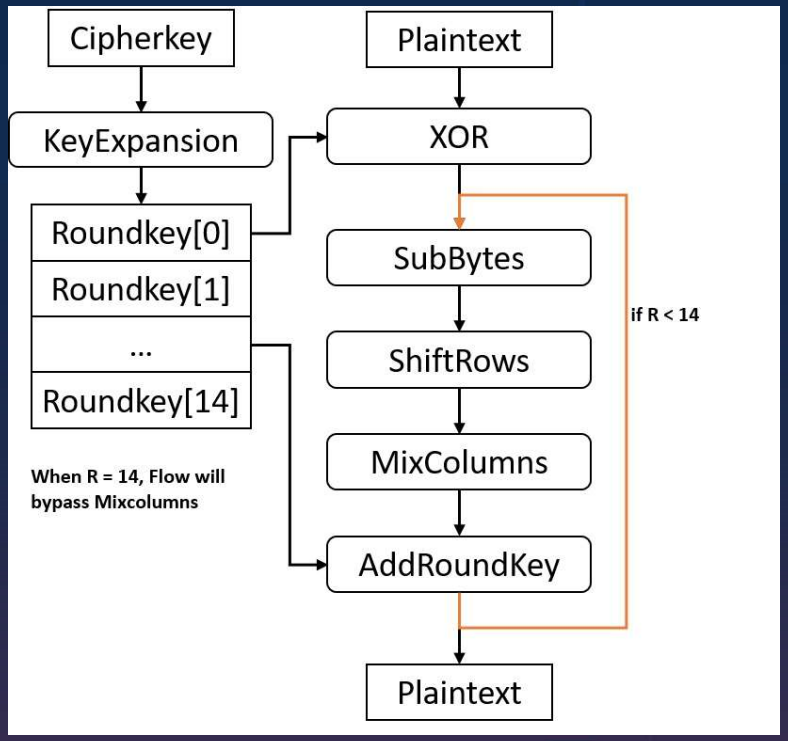
由于双射&混淆扩散，定义域整体无法化简。

每个映射都能很容易求逆。

MixColumns 双射存在与指令块级别，而不是指令级别，指令级别逆推不可。（四元一次方程组）

https://xilinx.github.io/Vitis_Libraries/security/2020.1/_images/original_flow.png

保护 MixColumns



(d, x, Q)1 = (汇编代码，分块方案，分块方案的集合)

代码多态膨胀，等价乱序，逆向检查边界模糊情况

(d, x, Q)2 = (输入输出数据，影响的局部性的表现形式，全集)

输入输出乱序，内插入一段大规模简单仿射变换

仿射变换就是 y = ax + b (注意做到有限域上)

https://xilinx.github.io/Vitis_Libraries/security/2020.1/_images/original_flow.png

自定义 AddRoundKey

映射缺陷：指令级别可逆性

信息缺陷：指令没有歧义性，数据没有混淆扩散，key expansion名算法太显眼

指令级别可逆性 -> 二输入门的使用（NAND, NOR, IMPLY）——真实案例

数据没有扩散 -> 大矩阵乘法 ——真实案例

指令没有歧义性 -> 虚拟机 ——太常见了

NOR！！！

Vmp

于是要提到，Vmp充满歧义性的vAdd

点评一下VMP

(d, x, Q) = (内存，设计层面输入输出的局部划分，全集)

栈上操作

一级结构——汇编指令——局部栈操作

二级结构——数据访问的段落性——信息被擦除

(d, x, Q) = (断点信息/类windbg r，当前状态，全集)

寄存器轮转

context的数据的歧义性增加，状态更难获得

编译过程的解耦合

然而在CTF中并不常见

handler 体系的保护

Themida是handler体系的代表，虽然虚拟机都会用这个体系。

handler体系面临的威胁：

handler本身的不可区分性不佳

套娃方法造成的实质性复杂度提升不佳，高于二次套娃意义不大（复杂度/体积，归约证明）

二次套娃的意义：

handler之handler比真实运算的handler更难揣测其意义

handler的明显边界（跳转）被模糊。

Tips：handler多态，伪handler，随机选取

方法具有通用性，在AES算法保护中，优势并不明显。（归约）

控制流保护

之前提到“等价乱序”是线程流程下的。这里的控制流主要是条件控制流。

条件控制流在算法层面不容易等价乱序。

用花指令打破连贯性

攻：被patch

防：自校验

（如果攻防知道这就是花指令，过校验很简单，但谁知道这个自校验是不是真实算法的一部分呢）

花代码块：

永假条件。运行时加解密。（模拟执行显特征）

控制流保护之CFF

CFF(Control Flow Flattening)是比较知名的抗静态分析的代码混淆技术。

就 if 和 while 给出示例：

```
while (loop){
  switch (state){
  case INIT:
    state = select_S1_S2(...);
    break;
  case S1:
    do job1; state = W;
    break;
  case S2:
    do job2; state = W;
    break;
  ...
  case END: loop = false;
break;
  }
}
```

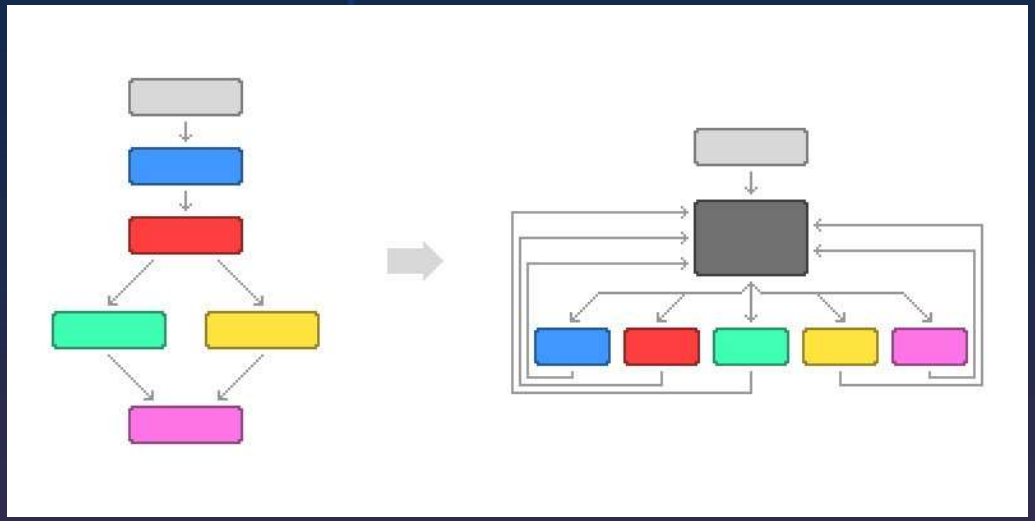
```
while (loop){
  switch (state){
  ...
  case W:
    do job3;
    state = select_W_WNEXT(...);
    break;
  case WNEXT:
    ...
  ...
  }
}
```

控制流保护之CFF

CFF 把程序流程变为平坦模式，AST静态特征被擦除，动态跳转。

概念：基本块，真实块，虚假块

通过模拟执行，符号执行等，识别真实/虚假块，还原真实流程。



https://blog.jscrambler.com/jscrambler-101-control-flow-flattening

控制流保护优化

攻击方需要解决的问题：

(跳转指令，代码分块，全集) -> 编号

(代码块执行序列，基本块真假，全集) -> 化简

(代码块执行序列，基本块结构，全集) -> 拉直

(数据访问，基本块分组，全集) -> 操作特定数据的基本块与特定算法相关

防守方让问题更难：

抗代码块识别：基本块插入花指令，跳转指令冗余，范围跳转随机化。（我们无法隐藏CFF基本架构，但我们可以让初步分块更复杂）

抗化简：跑了等于没跑的虚假块比根本跑不到的虚假块更好

执行序列与输入相关

数据组织与访问冗余

控制流保护优化

控制流具有设计层面的确定性。我们能否让控制流与用户输入相耦合?

一种容易实现的方法:

用户输入=(算法输入，校验数据)， 校验数据与算法输入相匹配。

算法输入与校验数据共同决定基本块执行顺序，不同的算法输入对应的序列不同。

（假定，仅靠示例输入难以还原校验算法）

攻击手法:

校验为避免多解，往往要在校验数据错误时把目标结果排除于值域之外。

否则算法和校验耦合严重，难以设计。

通过值域分析，把值域包含目标结果作为约束，可以从局部还原校验算法。

问题，有没有通解?

控制流保护优化

归约证明：

假如我们用于校验的基本块有256个，代码如下：

```
global counter = 0;
SelfCheckBasicBlock_X(data){
  data[counter] = data[counter] xor X;
  counter++;
}
```

用此基本块序列来代替AddRoundKey.

这道题不可解！

门电路

NAND, NOR, IMPLY / NIMPLY 可以实现全部的运算指令，这样的逻辑门称为万用门。

容易构造自定义万用门。

指令由(变长)门电路块决定。

门电路块的任意分割在指令层面全同。

代码无分支，流程全同。

立即数可由连接方式代替。

门电路

原指令的不可区分性 -> NAND/NOR名设计 or 从数据入手

控制流的不可区分性 -> 全同

数据流的不可区分性 -> 门电路本身没有很好的消除数据流特征

敏感数据所在存储空间的不可区分性 -> 敏感数据在结构中

方案Tips:

避开名设计 -> 自定义门电路 / 基于控制位的多态门

保护数据流 -> 数据操作扫描化/随机化（事实上，CTF中极力保留特征以降低难度）

保护数据流 -> 非线性算法（混淆） & 数据扩散

保护敏感数据 -> 算法耦合（算法复合化简）

一种实践成功的套路

可逆逻辑门 Reversible Gate

RGX (Reversible Gate X，基于经典的逻辑门RUG)

ABC是输入，PQR是输出，RGX运算规则如下：（+ 代表 xor）

P=AB+!A!C

Q=AB+BC+CA

R=B!C+!BC

SOP (Sum of Products) 对应于真值表

我们把标准 SOP 的 or 改成了我们使用的 xor，性质不变

输出位 = non-cannonical SOP = cannonical SOP

Q = AB+BC+CA = ABC + !ABC + A!BC+ AB!C

一种实践成功的套路

大规模随机SOP的爆破难度等同于建立真值表（指数）

用 RGX 与大矩阵乘法符合，结果化简，制造拟随机大规模SOP

采用不同带宽和逻辑的RG可以更好的拟随机大规模SOP

RGX自身实现混淆边界、欺骗IDA

采用二元门电路实现RGX

多态实现

比较指令愚弄IDA

名黑盒方法检查

更深的套路展望

Barrington定理

五阶矩阵乘法可实现：and和not

用is_zero判断0和1

同态加密套路

GGH

虽然这个是可以破解的，但已经不是正常CTF题的难度了

更深的套路展望

ABC	P=(AB)^(!A!C)
000	1
001	0
010	1
011	0
100	0
101	0
110	1
111	1

```
(base) E:\projs\
(0, 0, 0) 1
(0, 0, 1) 0
(0, 1, 0) 1
(0, 1, 1) 0
(1, 0, 0) 0
(1, 0, 1) 0
(1, 1, 0) 1
(1, 1, 1) 1
```

```
,0,1,0,0],[0,0,0,1,0],[0,0,0,0,1]])
0,0,1,0,0],[0,1,0,0,0],[0,0,0,0,1]])
0,0,0,1,0],[0,1,0,0,0],[0,0,0,0,1]])
0,1,0,0,0],[0,0,0,1,0],[0,0,0,0,1]])
0,0,0,1,0],[0,1,0,0,0],[0,0,0,0,1]])
0,0,1,0,0],[0,1,0,0,0],[0,0,0,0,1]])
1,0,0,0,0],[0,1,0,0,0],[0,0,0,0,1]])
```

```
.I)*(((A1.I*A)*(C1.I*C)*(A1.I*A).I* \
*((A*B*A.I*B.I)*M1.I).I*(((A1.I*A)* \
.I*(C1.I*C).I)*M2.I).I*M3.I
ult == Z).all() else 1)
```

总结

多态膨胀。等价乱序。

花指令，伪代码块，永假条件。

跳转指令等价替换、冗余和移除。动态跳转。控制流平坦化。随机跳转。

运行时加解密。自校验。

避免单变量映射，大规模方程组化，矩阵化。扩散。

经典万用门。可逆计算逻辑门。扩展万用门（控制位）。Barrington定理。

虚拟机架构，handler逻辑模糊，handler复用。状态机。

（不建议）用户输入参与的控制流。

检验多解。检验值域特征。检查边界模糊情况。试试黑盒名算法。

谢谢

看雪ID: 大龙猫