“强智算法”算法分析报告

第 2 小组

在报告开始前我们需要指出，第 3 小组给出的算法实现有一定的漏洞，在处理某些输入时程序会产生异常退出，这在某种程度上影响了我们的分析过程，详见附录中的环境说明。

1 Introduction

在这篇报告中，我们对第 3 小组的“强智算法”进行分析。在第 2 节，我们使用 NIST 随机性检测工具对其进行随机性检测。在第 3 节，我们首先尝试进行区分攻击，然后分别使用 Nivasch 算法和 Sedgewick 两种算法对该算法进行碰撞攻击。在第 4 节，我们详细地分析了该算法中不同部件的作用。在第 5 节，我们对该算法的安全性进行了总体性的评估。

分析时所用代码见仓库地址：https://github.com/Ashitemaru/Crypto-LAB3

2 随机性检测

我们使用 NIST 随机性检测工具，对该哈希函数的输出进行随机性检测。检测结果表明，该算法的随机性总体十分良好，对高密、低密和随机的输入数据，都能产生随机性较强的输出。

2.1 测试方法

2.1.1 数据输入

输入数据分为三种类型：

类型	特征
高密	每位取 1 的概率为 95%
低密	每位取 1 的概率为 5%
随机	每位取 1 的概率为 50%

对于每种类型的输入数据，我们生成长度为 800 bit 的数据流计算其哈希值，然后将哈希值拼接进行检测。

2.1.2 检测规格

- 128 B
- 16 KB

NIST 的部分检测对序列长度要求较高，

检测	要求
Binary Matrix Rank Test	$n \geq 38912$
Overlapping Template Matching Test	$n > 10^6$
Maurer's "Universal Statistical" Test	$n \geq 387840$
Linear Complexity Test	$n \geq 10^6$
Random Excursions Test	$n \geq 10^6$
Random Excursions Variant Test	$n \geq 10^6$

为了能尽可能的利用 NIST 提供的多种测试，我们增加一组检测规格。

- 128 KB

共进行 $3 \times 3 = 9$ 组实验。对于每种检测规格，我们都从生成的哈希中取 300 个输入流进行检测。

对于不符合测试参数的测试点，在后续分析的过程中予以剔除，其余测试点成称为有效测试点。

2.2 测试结果

以下测试的有效测试点全部通过。

- BlockFrequency
- CumulativeSums
- Frequency
- LinearComplexity
- LongestRun
- OverlappingTemplate
- RandomExcursions
- RandomExcursionsVariant
- Rank
- Runs
- Serial
- Universal

以下测试的部分测试点存在异常。

2.2.1 ApproximateEntropy

在数据规模较大时存在异常。

数据规模	输入类型	通过率	P值
16KB	高密	291/300	0.000082*
128KB	随机	289/300*	0.000012*

2.2.2 FFT

在 $n = 128B$ 时分布 P 值异常。

数据规模	输入类型	通过率	P值
128B	高密	297/300	0.000000*
128B	低密	293/300	0.000000*
128B	随机	297/300	0.000000*

2.2.3 NonOverlappingTemplate

Non-Overlapping Template Test 有很多子测试模板，对每种 (数据规模，输入类型) 组合，统计不同模板的测试通过比例和P值均一性测试通过比例。

在 $n = 128B, n = 128KB$ 时通过率较低，在 $n = 16KB$ 通过率较好。

数据规模	输入类型	检测通过比例	P值通过比例
128B	高密	42/148	0/148
128B	低密	42/148	0/148
128B	随机	41/148	0/148
16KB	高密	148/148	148/148
16KB	低密	148/148	148/148
16KB	随机	147/148	148/148
128KB	高密	113/148	139/148
128KB	低密	123/148	134/148
128KB	随机	119/148	129/148

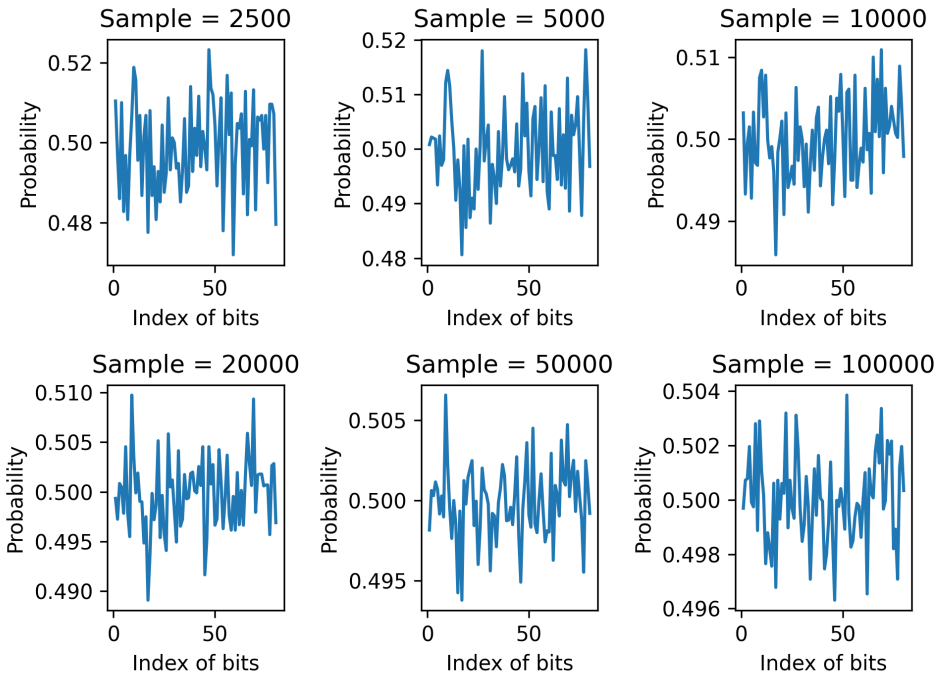
全部测试数据据详见附录。

3 安全性分析

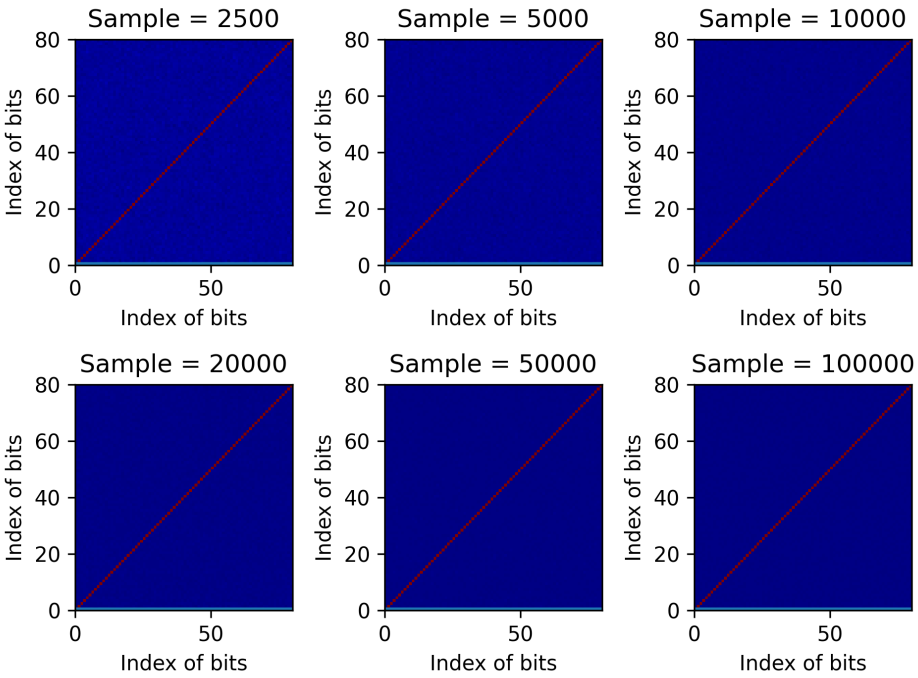
3.1 区分攻击

该算法对于区分攻击具有较强的抗性。我们使用该算法随机生成了 {2500, 5000, 10000, 20000, 50000, 100000} 个哈希值，然后对这些哈希值进行统计分析。

首先是逐 Bit 分析，我们统计每个 Bit 的 “1” 的出现次数。如果是随机分布，每个 Bit 应服从 $p = 0.5$ 的伯努利分布。绘出下图，证明其确实具有较好的随机性。测试脚本见 `./distinguishing_attack` 目录。



然后是检验不同 Bit 之间的关联性。我们绘制出下图，图中每个点的横纵坐标确定哈希值两个比特位，其数值代表两个比特位相同的概率。可以看出不同比特之间没有明显的关联性。



3.2 碰撞攻击

3.2.1 多栈的 Nivasch 算法

碰撞攻击是说，是否存在 M_1，M_2，使得 $M_1 \neq M_2$ 但 $H(M_1) = H(M_2)$。我们使用多栈的 Nivasch 算法（G. Nivasch, "Cycle detection using a stack", Information Processing Letters 90/3, pp. 135-140, 2004.）进行实验。Nivasch 算法在被搜索函数 H 的值域 D 建立偏序关系，通过寻找环上的最小元素来检测环的存在，当第 2 次经过环上最小元素时即可判断环的大小。测试脚本见

`./nivasch` 目录。

我们将算法实现中的 `Hash` 结构体视为一个大端整数建立偏序关系。给定初始串 M_0，记 $M_i = H(M_{i-1})$，由于值域有限，这条试探链中必然存在一个环。我们利用 Nivasch 算法确定环的长度，再结合长度信息用双指针找到环的入口，从而确定碰撞的原象。

由生日攻击的相关理论我们可以知道，对于一个有 2^n 个不同元素的集合，如果从中随机选取超过 $2^{\frac{n}{2}}$ 个元素，就有与 0.5 数量级相同的概率产生重复。而如果哈希算法的随机性足够高，那么当计算量达到 $2^{\frac{n}{2}}$ 量级时，就有较高概率找到相应的碰撞。

3.2.1.1 实验结果

我们对不同长度的 Hash 串进行了碰撞攻击。长度为 n bit 表示截取该哈希函数的前 n bit 构成新的值域 D 进行搜索。

长度	理论复杂度	找到碰撞所用迭代次数	圈长
8 bit	2^{4}	$2^{4.24}$	$2^{2.32}$
16 bit	2^{8}	$2^{8.60}$	$2^{7.74}$
24 bit	2^{12}	$2^{11.94}$	$2^{10.01}$
32 bit	2^{16}	$2^{16.47}$	$2^{14.94}$
40 bit	2^{20}	$2^{17.68}$	$2^{15.96}$
48 bit	2^{24}	$2^{23.54}$	$2^{22.29}$
56 bit	2^{28}	$2^{27.75}$	$2^{27.43}$
64 bit	2^{32}	$2^{32.65}$	$2^{31.62}$
72 bit *	2^{36}	$> 2^{37}$	N/A
80 bit *	2^{40}	$> 2^{37}$	N/A

* 表示未能找到碰撞。

由于算力有限，我们没有找到在 72 bit 和 80 bit 下该哈希函数的碰撞。从已知的碰撞结果来看，该算法在不同长度的碰撞攻击下表现良好，随机路径迭代复杂度在大部分情况下与理论值接近。

实验中找到的碰撞值，算法初值，Nivasch 算法的中间结果详见附录。

3.2.2 Sedgewick 算法

Reference: Sedgewick, Robert; Szymanski, Thomas G.; Yao, Andrew C.-C. (1982), "The complexity of finding cycles in periodic functions", SIAM Journal on Computing, 11 (2): 376–390, doi:10.1137/0211030

Link: https://epubs.siam.org/doi/10.1137/0211030

Sedgewick，Szymanski，Yao 提出了一种算法（下面简称 Sedgewick 算法），可以在使用 M 内存空间的情况下，用最坏情况下 $(\lambda+\mu)(1+\frac{c}{\sqrt{M}})$ 次的函数计算找到环。与 Nivasch算法 相比，虽然 Sedgewick 算法的最坏时间复杂度要高于 Nivasch 算法的平均复杂度，但是要远低于 Nivasch 算法的最坏复杂度。由于哈希函数的环可能会非常长，因此我们认为有测试 Sedgewick 算法的必要性。

设哈希函数为 $f(x)$，其迭代 n 次的值为 $f^n(x)$。暴力算法会储存所有的 $(f^i(x),i)$ 来检测是否存在环，Sedgewick 算法使用了大小上限为 M 的表 `TABLE` 来存储这些值，插入的的值满足 $(f^{kb}(x),kb),k\in\mathbb{Z}$，搜索的区间为 $i\in[0,g\cdot b)$，如果搜索到对应的在 `TABLE` 内的值，那么证明检测到环。`TABLE` 肯定无法储存所有的函数值，但是如果检测到它是满的，可以倍增 b 的值让其当前大小减半，这也相当于扩大环长度的搜索范围。这个算法保证可以检测到环并退出，因为如果存在环，那么连续的长度为b的搜索区间一定能遇到 $f^{kb}(x)$。关于确切的最差情况复杂度证明请参考原论文。

g 是一个固定参数，通过合理调整它的值，可以优化其理论总运行时间到 $O(\sqrt{1/M})$ 的级别。更确切的说，我们有：

$$g=\frac{Mt_s}{16t_f}(1+\frac{14}{M})$$

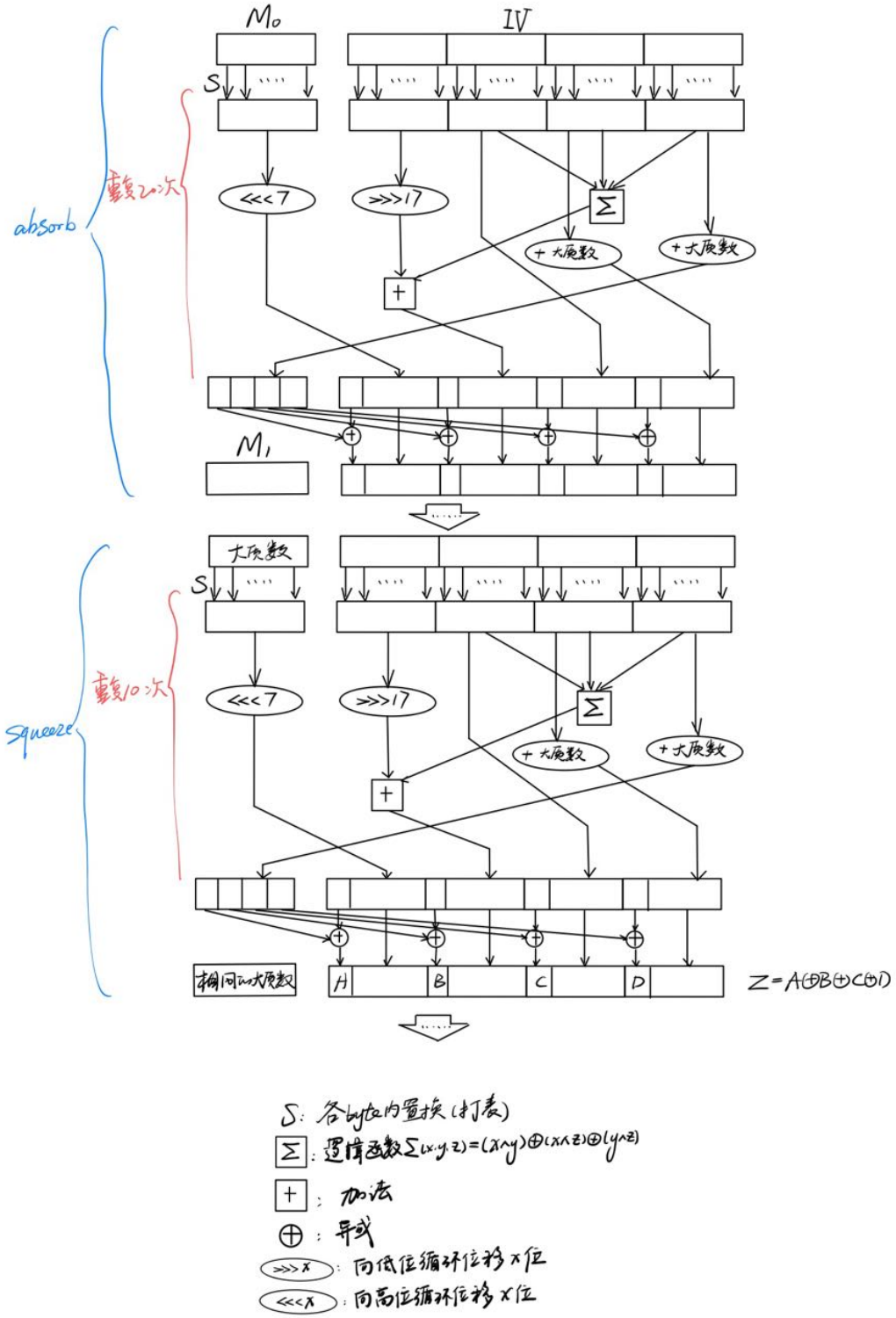
其中 t_s,t_f 分别为在 `TABLE` 内搜索和查找值所需时间。`TABLE` 的实现可以多种多样，其时间复杂度可以是 $O(1)$，$O(\log M)$，甚至可以是线性查找表的 $O(M)$。在我们的实现中，使用了 `std::unordered_map` 提供的均摊时间复杂度为 $O(1)$ 的哈希表。这样我们可以直接计算出最优化的 $g=\frac{M}{16}(1+\frac{14}{M})$。

检测到存在环后，我们实际上得到的是环的长度 c。恢复时，首先计算环的最小可能起点 j'，然后在向后寻找最小的 $l>j'$，使得 $f^l(x)=f^{l+c}(x)$，这样我们就得到了一个碰撞：$f(f^{l-1}(x))=f(f^{l+c-1}(x))$。这个过程中需要计算 $f^{j'}(x)$，为了加速过程，我们可以利用表中已有的信息，寻找 $f^{b\lfloor j'/b\rfloor}(x)$，然后再进行 $j'\bmod b$ 次函数计算即可求得 $f^{j'}(x)$。具体计算方法参见论文和代码实现。

综上，Sedgewick 算法可以在使用 M 大小的内存空间的情况下，用 $n(1+\Theta(1/\sqrt{M}))$ 步操作找到一个碰撞，即第一个重复的 $f^n(x)$。在我们的实现中，取 $M=200000000$，稳定运行占用内存大约有 10GB，能在 17s 内找到 6 字节的碰撞，500s 左右找到 7 字节碰撞。对于更高字节数的碰撞，由于计算资源和时间问题，还未找到。根据估计，对于 9 字节碰撞，可以在 150 小时内完成。测试脚本见 `./sedgewick` 目录。

4 Ablation Study

在这一部分我们尝试分析算法中每一个部件的作用，以尝试解释其产生的结果。同时，根据我们对于部件作用的分析，我们会给出为了保证算法整体的安全性，每个部件至少所应选取的参数范围。下图为“强智算法”报告中的实现流程图，我们按照从上到下的顺序依次分析。



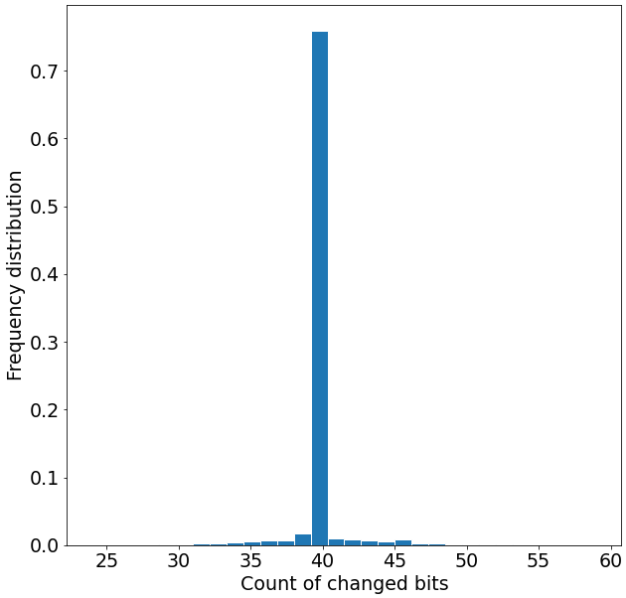
4.1 S-Box

首先是 S-Box，这个置换起到的作用是，将输入字符中“邻近”的特性均匀地混淆。也就是说，即使 S-Box 的输入 I_1, I_2 仅仅有 1 Bit 的不同，经过 S-Box 的置换之后所产生的输出结果 $S_1 = S(I_1), S_2 = S(I_2)$ 是完全不同的。这里置换表在算法给出时已经固定。

4.1.1 扩散测试

- 随机给定一个输入串 M，计算哈希值 H_1。
- 对串 M 选取一位进行翻转，然后再计算哈希值 H'
- 计算 H 和 H' 中不同有位数的数量 B_i

取长度为 8×10^4 的 M，逐位翻转，结果如下。



对结果定量计算

1. 改变位数的最小值 $B_{\min} = \min\left(\{B_i\}_{i=1,\ldots,N}\right)$
2. 改变位数的最大值 $B_{\max} = \max\left(\{B_i\}_{i=1,\ldots,N}\right)$
3. 改变位数的标准差 $\Delta B = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}\left(B_i - \bar{B}\right)^2}$
4. 改变位数的平均值 $\bar{B} = \frac{1}{N}\sum_{i=1}^{N} B_i$
5. 每位的改变概率 $P = \left(\frac{\bar{B}}{80}\right) \times 100\%$

最小值	最大值	标准差	平均值	每位变化概率
24	59	1.61	40.00	50.00%

得益于 S-Box 作用，该算法在扩散测试中表现十分理想，其在扩散测试中的分布与典型的正态分布截然不同。

4.2 Absorbing

为了方便我们分析 Absorbing 的轮函数的效果，我们选取了特殊的输入 I_1, I_2，使得 $S_1 = S(I_1), S_2 = S(I_2)$ 的海明距离只有 1 Bit。也就是说，我们从 S 盒中逆向寻找了输入 I_1, I_2。为了探究 Absorbing 轮函数的作用轮数对 Absorbing 阶段输出结果的影响，我们做了以下探究实验：

选取输入 $I_1 = \{$0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x00$\}$，输入 $I_2 = \{$0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x21$\}$，这里 S_1, S_2 只有 1 Bit 的差距。我们使用该串作为 Hash 函数的初始输入，输出在做长度填充之前的隐藏状态 state 的字节表示。

Absorb 轮数	I_1 的输出	I_2 的输出
1	C988397AC302B838 **21B40**DE2D516AFCB 48967973C0450B40 167FCC7F0903FADD	C988397AC302B838 **22008**DE2D516AFCB 48967973C0450B40 167FCC7F0903FADD
2	FE99A96DCB42F3ED **9F4FDC**DE2DEC1D04 **1F5C46**43A6510D5A **25EA79**CBBA8388E9	FE99A96DCB42F3ED **6F6FED**DE2DEC1D04 **C77E79**43A6510D5A **F97F47**CBBA8388E9
3	**C4C0F**26DDBA2**08F1** **2DDB50**0F50D**EC353** E**1B106**93CE5D**943**E **EC3C6355E39A62CB**	**7EC07**26DDBA2**E4CD** **3FD4FC**0F50D**D3F5B** E**0AAA2**93CE5D**563**E **1B051**55E39A**49C2**
4	**880BC5**A0C11**D224C** **BE3C69BE735E17D7** **2B9466D659B537**AA **EAE3CA**786BF49**38A9**	**253845**A0C1**7AF1C6** **1E72CC3355D3683** **2660888497B5A5**AA **EFA76**786BF4**E680**
8	**8438F56A76C25C4B** **393A367140C89135** **802754ABD2142B50** **A2EC45E40D0F7060**	**44732320CA953760** **EEDEF58B08612B08** **423B0245C458EB12** **FB5ABEDB4D992575**
20	**D8ABA80C2461B192** **6AFBA15F96E13AAE** **3227708359202BD** **E06D28DDE78D6DC3**	**50074C4626DA1E7C** **F5816A7E44D523FC** **D307A9D5EE9A8799** **872FEB1B5C4D7033**

根据实验结果可以看到，在经过 4 轮迭代后，1 个 Bit 的影响基本消除，而经过 8 轮迭代后，则完全产生了雪崩效应。这里原算法给出的吸收轮数为 20，效果与 8 轮基本相同。因此，该算法若要达到安全性，吸收的轮数至少为 8 轮。

4.3 Squeezing

我们依然选取相同的 I_1 和 I_2，Absorbing 阶段设置为 1 轮，探究 Squeezing 阶段迭代轮数对输出结果的影响。

Squeeze 轮数	$H(I_1)$	$H(I_2)$
1	65 43 **3B BF A0 04 12 12 EE 7D**	65 43 **0D C4 86 E7 C1 CB CD 71**
2	**EF 44 2F 43 35 65 84 F3 60 A8**	**D9 3F EE 63 06 D9 A1 0C 86 FB**
3	**68 AE BC DA 04 83 70 27 65 B2**	**69 4D C0 D1 A6 BC FC A5 B6 CB**
10	**6E FF 8A 3D 26 92 FF DC FD B6**	**00 AA 5B 9F 71 ED AA 12 3B E6**

得益于每次 Squeeze 只取 16 Bit 的特性，在 Squeezing 阶段的输入的差分并不大时，其前两个字节会有明显碰撞，而其后的输出结果均均匀扩散，这也得益于大质数的影响。在 Squeezing 阶段迭代轮数增加到 2 时，输出结果便得到了完全均匀的扩散。原算法这里给出的挤出迭代轮数为 10。而我们提出，要想达到安全性，挤出的轮数至少为 2 轮。

5 总结

在本篇分析报告中，我们分析了第 3 组给出的算法“强智算法”。他们的设计理念是：“基于 Sponge 结构进行变形，缝合 MD5、SHA-3 等算法的优秀设计，大量使用查表置换、位移、加法、异或、逻辑函数等运算使轮函数尽可能随机”。综合分析与评判后，我们认为该哈希函数算法具有较强的安全性。

我们首先对该算法进行了随机性检测，检测结果表明该算法的随机性十分良好。然后，我们尝试对该算法进行攻击，包括区分攻击与（缩短长度的）碰撞攻击。统计结果与攻击结果均表明，该算法相对于其结果的位数而言，是计算安全的。然后，我们尝试分部件对该算法的安全性进行分析，旨在找出该算法中是否有部件存在明显的安全漏洞。经逐步试探检验（减轮攻击），我们并没有发现该算法有明显的安全漏洞，并通过分析过程，给出了要想让该算法达到安全的超参数的实践下界。

最后，感谢课程组对本次作业的指导，这是对于课上所学到的密码分析技术的一次有效实践。

6 附录

6.1 测试环境

以上的分析中，我们使用的编译命令为 `g++ * -O3 main`，测试环境为：

```
WSL2 Ubuntu20.04
i5-9300H @ 2.4GHz
g++ 9.4.0
xmake 2.6.6
```

6.2 实验框架介绍

```
.
├── README.md                # 实验报告
├── README.pdf
├── ablation                 # Ablation Study
├── collision_attack         # 基于集合的碰撞攻击
├── diffusion                # 扩散测试
├── distinguishing_attack    # 区分攻击
├── nivasch                  # 基于 Nivasch 算法的碰撞攻击
├── sedgewick                # 基于 Sedgewick 算法的碰撞攻击
└── random_analysis          # 随机性测试
```

6.3 随机性检测详细结果

6.3.1 测试框架说明

本项测试的代码与数据见 `./random_analysis` 目录

- 每项测试的原始报告及测试脚本见 `128B`，`16KB`，`128KB` 三个目录。
- 测试时使用的随机数生成程序见 `main.c`
- 将原始测试结果转化成如下表格的工具为 `report_gen.ipynb`

下面对每类测试的通过情况进行列举，通过率不达标或 P 值过低的，用 $*$ 进行标注。

6.3.2 ApproximateEntropy

以下测试点存在异常：

数据规模	输入类型	通过率	P值
16KB	高密	291/300	0.000082*
128KB	随机	289/300*	0.000012*

其余测试点全部通过：

数据规模	输入类型	通过率	P值
128B	高密	295/300	0.487885
128B	低密	300/300	0.798139
128B	随机	297/300	0.699313
16KB	低密	293/300	0.000555
16KB	随机	295/300	0.001943
128KB	高密	293/300	0.000427
128KB	低密	294/300	0.000111

6.3.3 BlockFrequency

有效测试点全部通过。

数据规模	输入类型	通过率	P值
128B	高密	295/300	0.487885
128B	低密	297/300	0.561227
128B	随机	299/300	0.090936
16KB	高密	299/300	0.383827
16KB	低密	295/300	0.075719
16KB	随机	297/300	0.401199
128KB	高密	299/300	0.616305
128KB	低密	295/300	0.013889
128KB	随机	297/300	0.726503

6.3.4 CumulativeSums

有效测试点全部通过。

数据规模	输入类型	通过率	P值
128B	高密	296/300	0.746572
128B	高密	297/300	0.003046
128B	低密	296/300	0.150906
128B	低密	298/300	0.047682
128B	随机	296/300	0.000682
128B	随机	295/300	0.006048
16KB	高密	299/300	0.366918
16KB	高密	298/300	0.087338
16KB	低密	297/300	0.637119
16KB	低密	298/300	0.798139
16KB	随机	300/300	0.872947
16KB	随机	299/300	0.990369
128KB	高密	295/300	0.345115
128KB	高密	294/300	0.678686
128KB	低密	299/300	0.228764
128KB	低密	299/300	0.443451
128KB	随机	298/300	0.168733
128KB	随机	297/300	0.791880

6.3.5 FFT

以下测试点存在异常：

数据规模	输入类型	通过率	P值
128B	高密	297/300	0.000000*
128B	低密	293/300	0.000000*
128B	随机	297/300	0.000000*

其余测试点全部通过：

数据规模	输入类型	通过率	P值
16KB	高密	292/300	0.994250
16KB	低密	294/300	0.425059
16KB	随机	297/300	0.931952
128KB	高密	295/300	0.195163
128KB	低密	296/300	0.009311
128KB	随机	291/300	0.657933

6.3.6 Frequency

有效测试点全部通过。

数据规模	输入类型	通过率	P值
128B	高密	297/300	0.862344
128B	低密	298/300	0.534146
128B	随机	296/300	0.014550
16KB	高密	297/300	0.217094
16KB	低密	297/300	0.609377
16KB	随机	300/300	0.822534
128KB	高密	295/300	0.032203
128KB	低密	300/300	0.289667
128KB	随机	297/300	0.872947

6.3.7 LinearComplexity

有效测试点全部通过。

数据规模	输入类型	通过率	P值
128KB	高密	296/300	0.602458
128KB	低密	296/300	0.554420
128KB	随机	296/300	0.785562

6.3.8 LongestRun

有效测试点全部通过。

数据规模	输入类型	通过率	P值
128B	高密	299/300	0.692455
128B	低密	299/300	0.202268
128B	随机	296/300	0.129620
16KB	高密	298/300	0.872947
16KB	低密	297/300	0.220931
16KB	随机	298/300	0.419021
128KB	高密	294/300	0.195163
128KB	低密	299/300	0.946308
128KB	随机	297/300	0.816537

6.3.9 NonOverlappingTemplate

Non-Overlapping Template Test 有很多子测试模板，在这里不进行逐一列举。

对每种 (数据规模，输入类型) 组合，考察测试通过率和P值均一性测试通过率。

数据规模	输入类型	检测通过比例	P值通过比例
128B	高密	42/148	0/148
128B	低密	42/148	0/148
128B	随机	41/148	0/148
16KB	高密	148/148	148/148
16KB	低密	148/148	148/148
16KB	随机	147/148	148/148
128KB	高密	113/148	139/148
128KB	低密	123/148	134/148
128KB	随机	119/148	129/148

6.3.10 OverlappingTemplate

有效测试点全部通过。

数据规模	输入类型	通过率	P值
128KB	高密	299/300	0.115387
128KB	低密	294/300	0.165646
128KB	随机	297/300	0.893001

6.3.11 RandomExcursions

有效测试点全部通过。

数据规模	输入类型	通过率	P值
128KB	高密	194/200	0.296834
128KB	高密	196/200	0.946308
128KB	高密	200/200	0.834308
128KB	高密	198/200	0.851383
128KB	高密	198/200	0.202268
128KB	高密	198/200	0.224821
128KB	高密	197/200	0.524101
128KB	高密	199/200	0.709558
128KB	低密	188/194	0.196086
128KB	低密	190/194	0.461912
128KB	低密	193/194	0.130014
128KB	低密	193/194	0.095365
128KB	低密	191/194	0.040385
128KB	低密	193/194	0.856619
128KB	低密	192/194	0.598820
128KB	低密	191/194	0.587927
128KB	随机	169/170	0.877806
128KB	随机	170/170	0.018652
128KB	随机	168/170	0.194135
128KB	随机	169/170	0.025698
128KB	随机	168/170	0.679903
128KB	随机	166/170	0.379806
128KB	随机	170/170	0.102885
128KB	随机	168/170	0.903500

6.3.12 RandomExcursionsVariant

有效测试点全部通过。

数据规模	输入类型	通过率	P值
128KB	高密	200/200	0.626709
128KB	高密	199/200	0.105618
128KB	高密	197/200	0.167184
128KB	高密	197/200	0.289667
128KB	高密	197/200	0.428095
128KB	高密	197/200	0.289667
128KB	高密	195/200	0.474986
128KB	高密	197/200	0.242986
128KB	高密	198/200	0.176657
128KB	高密	200/200	0.167184
128KB	高密	198/200	0.595549
128KB	高密	200/200	0.534146
128KB	高密	199/200	0.668321
128KB	高密	199/200	0.935716
128KB	高密	199/200	0.749884
128KB	高密	199/200	0.719747
128KB	高密	199/200	0.358641
128KB	高密	199/200	0.825505
128KB	低密	193/194	0.534146
128KB	低密	192/194	0.642599
128KB	低密	191/194	0.403984
128KB	低密	192/194	0.422829
128KB	低密	193/194	0.534146
128KB	低密	193/194	0.118630
128KB	低密	192/194	0.231703
128KB	低密	193/194	0.073776
128KB	低密	193/194	0.544788
128KB	低密	189/194	0.577070
128KB	低密	187/194	0.359074
128KB	低密	189/194	0.098423
128KB	低密	191/194	0.231703
128KB	低密	192/194	0.653554
128KB	低密	193/194	0.225440
128KB	低密	193/194	0.555494
128KB	低密	194/194	0.781206

数据规模	输入类型	通过率	P值
128KB	低密	194/194	0.598820
128KB	随机	169/170	0.099338
128KB	随机	169/170	0.313747
128KB	随机	169/170	0.818661
128KB	随机	169/170	0.049645
128KB	随机	168/170	0.071961
128KB	随机	169/170	0.570068
128KB	随机	168/170	0.182140
128KB	随机	169/170	0.369867
128KB	随机	168/170	0.360093
128KB	随机	167/170	0.296409
128KB	随机	169/170	0.961917
128KB	随机	169/170	0.582174
128KB	随机	169/170	0.546044
128KB	随机	169/170	0.188060
128KB	随机	169/170	0.379806
128KB	随机	169/170	0.083167
128KB	随机	169/170	0.630995
128KB	随机	169/170	0.453721

注：RandomExcursion及其变种有包含多个测试项目。

6.3.13 Rank

有效测试点全部通过。

数据规模	输入类型	通过率	P值
16KB	高密	294/300	0.999438
16KB	低密	297/300	0.437274
16KB	随机	298/300	0.425059
128KB	高密	298/300	0.224821
128KB	低密	296/300	0.481416
128KB	随机	295/300	0.249284

6.3.14 Runs

有效测试点全部通过。

数据规模	输入类型	通过率	P值
128B	高密	298/300	0.935716
128B	低密	295/300	0.209577
128B	随机	297/300	0.588652
16KB	高密	298/300	0.474986
16KB	低密	296/300	0.581770
16KB	随机	299/300	0.964295
128KB	高密	296/300	0.949602
128KB	低密	298/300	0.753185
128KB	随机	297/300	0.162606

6.3.15 Serial

有效测试点全部通过。

数据规模	输入类型	通过率	P值
128B	高密	298/300	0.961593
128B	高密	293/300	0.153763
128B	低密	298/300	0.906970
128B	低密	299/300	0.228764
128B	随机	296/300	0.872947
128B	随机	299/300	0.425059
16KB	高密	294/300	0.096578
16KB	高密	295/300	0.507512
16KB	低密	299/300	0.245072
16KB	低密	298/300	0.581770
16KB	随机	294/300	0.798139
16KB	随机	297/300	0.014216
128KB	高密	297/300	0.455937
128KB	高密	298/300	0.228764
128KB	低密	298/300	0.010237
128KB	低密	296/300	0.113151
128KB	随机	297/300	0.350485
128KB	随机	298/300	0.657933

6.3.16 Universal

有效测试点全部通过。

数据规模	输入类型	通过率	P值
128KB	高密	297/300	0.766282
128KB	低密	299/300	0.345115
128KB	随机	296/300	0.520767

6.4 Nivasch 算法中间结果

下面列举了算法运行的原始结果

- Minimum Element 表示找到的最小元素
- Step 表示搜索的迭代次数
- Stack 表示最小元到起点的长度
- Cycle 表示圈的长度
- x_1，x_2 表示找到的碰撞。

6.4.1 初值

```
01 00 00 00 00 00 00 00 00 00
```

6.4.2 Len: 1 byte

Item	Value
Minimum Element	25
Steps	19
Stack	14
Cycle	5
x_1	25
$H(x_1)$	DD
x_2	86
$H(x_2)$	DD

6.4.3 Len: 2 byte

Item	Value
Minimum Element	00 04
Steps	389
Stack	175
Cycle	214
x_1	0C 2F
$H(x_1)$	4C D9
x_2	11 77
$H(x_2)$	4C D9

6.4.4 Len: 3 byte

Item	Value
Minimum Element	0C 33 54
Steps	3919
Stack	2888
Cycle	1031
x_1	D5 F2 B0
$H(x_1)$	F6 8A 26
x_2	EE AA 29
$H(x_2)$	F6 8A 26

6.4.5 Len: 4 byte

Item	Value
Minimum Element	00 59 62 46
Steps	90918
Stack	59565
Cycle	31353
x_1	2D 3E 65 8D
$H(x_1)$	8E BA A9 65
x_2	20 CA 43 0A
$H(x_2)$	8E BA A9 65

6.4.6 Len: 5 byte

Item	Value
Minimum Element	00 36 04 48 4E
Steps	210193
Stack	146254
Cycle	63939
x_1	13 96 40 57 6E
$H(x_1)$	04 FF 8D 22 DA
x_2	A2 61 99 06 D8
$H(x_2)$	04 FF 8D 22 DA

6.4.7 Len: 6 byte

Item	Value
Minimum Element	00 01 28 79 38 CD
Steps	12218212
Stack	7095393
Cycle	5122819
x_1	10 83 AE F0 83 24
$H(x_1)$	92 D7 DE 26 96 E5
x_2	69 30 72 58 51 C6
$H(x_2)$	92 D7 DE 26 96 E5

6.4.8 Len: 7 byte

Item	Value
Minimum Element	00 00 35 18 0A 5C AB
Steps	225887091
Stack	45179942
Cycle	180707149
x_1	3B 6D 42 68 BC 8D CE
$H(x_1)$	9B 16 B3 DA 8B 50 AB
x_2	0A 34 15 2B 29 AA 5E
$H(x_2)$	9B 16 B3 DA 8B 50 AB

6.4.9 Len: 8 byte

Item	Value
Minimum Element	00 00 47 01 48 03 D9 F1
Steps	6719350199
Stack	3407502794
Cycle	3311847405
x_1	F0 65 26 6C 94 F9 FA 77
$H(x_1)$	94 A8 51 76 52 85 56 23
x_2	87 87 E8 AF 34 68 8D 19
$H(x_2)$	94 A8 51 76 52 85 56 23

6.5 实现问题

我们指出算法实现有漏洞的地方在于，在处理某些文件时，其会产生异常退出。然而，对于其它文件，该算法实现不会产生该现象。在分析时，我们按照算法设计报告中的理念图，在重写了部分代码实现逻辑的基础之上撰写了我们的测试代码。