Exploiting Stream Cipher Key Reuse in Malware Traffic

September 3, 2014

Ben Herzog

1 Abstract

Rolling your own crypto can easily turn out to be a bad idea - this holds true when designing any software, and malware is no exception. We give a brief overview of the role of cryptography in malware communication, of the stream cipher cryptosystems often used in this capacity, and of those cryptosystems' linearity properties. We examine a scenario, based on an actual malware sample found in the wild, where a stream cipher is used repeatedly for encrypting different plaintexts with the key held fixed. We explain how this introduces a critical weakness to the cryptosystem due to the distinctive character distribution of any two xored ciphertexts. We briefly touch on the concept of Bayesian log-probability evidence bits, and introduce a naïve algorithm which utilizes this concept, taking advantage of the aforementioned weakness to recognize vulnerable ciphertext in time quadratic in the message length.

2 Encrypted Malware Traffic: Why and How

During the rise of personal computing in the '80s, once a piece of malware successfully infected a machine, the former was left to its own devices. It would report to no one and wait for no orders, executing its predefined functionality and no more. For the original author of the malware, it would in most cases be very difficult - if even possible - to ascertain that infection was successful or to somehow get the malware to follow a new set of instructions, not included in advance.

The rise of the World Wide Web put an end to that era. Nowadays, many pieces of malware communicate over the web in order to send reports, receive instructions, exfiltrate data and carry out many other modes of functionality. This 'noise' generated by malware has, in turn, led to the emergence of techniques attempting to intercept that traffic while in transit. These techniques generally employ approaches reliant on Deep Packet Inspection, such as protocol anomaly detection and pattern matching. The thing common to those kinds of techniques is that they must have access to the information being sent - some of it, if not all. If an algorithm cannot read and understand some structure in a packet, it certainly cannot rely on information gathered from that structure to decide whether the packet has malicious origins.

Malware authors have, subsequently, deployed various evasion techniques to get around agents attempting to detect the traffic generated by their creation. Some have turned to incorporating elements of steganography in malware communication - hiding information in plain sight, readily available for reading if only you know where to look. Other authors, instead, turned to cryptographic approaches. When an agent is looking at properly encrypted text, they should theoretically be unable to glean any information at all about what was being sent. This holds true for both a human observing the traffic and for any kind of automated algorithm doing the same.

The words 'properly' and 'theoretically' in that context carry a lot of weight. To wit:

- The level at which the encryption works might reveal any informative details present below that level (e.g. if you encrypt all HTTP payloads, it is still possible to mine all sorts of information from the HTTP header).
- Even at that same level where the encryption scheme operates, it might be possible to glean at least some basic facts such as the size of the data being sent.
- A human agent with access to the original malware might be able, with some effort, to reverse-engineer it and obtain the details of the encryption scheme, recovering secret keys and other information sufficient to break the encryption and read the plaintext.
- The encryption algorithm, or the details of the author's chosen implementation, may be vulnerable to attack.

That last scenario is particularly problematic from the malware author's point of view. In that case, the breaking of encryption is amenable to automation, does not require access to the malware bytecode, and in some cases, its success- in and of itself- raises significant suspicion regarding the traffic's nature. In this article, we will discuss a particular case of such a scenario: a cryptographic weakness that arises when stream cipher keys, or keystreams, are reused.

3 A Crash Course in One-Time Pads and Stream Ciphers

Though cryptography is a very old discipline- there are known uses of cryptography dating as far back as 1500 BCE - it was not until the 19th century that modern, rigorous notions of what a cryptosystem is, and what it means for a cryptosystem to be secure, began to develop. The era of classical cryptography saw many ciphers which in their time were thought to be unbreakable, which are in fact trivial to break with modern cryptanalytic techniques. Consider the Vigenère Cipher, a 16th-century invention; it withstood attacks for 3 centuries and was literally called "le chiffre indéchiffrable" (The unbreakable cipher) until in 1863 German cryptographer Friedrich Kasiski presented a general ciphertext-only total break of it - meaning, his algorithm could recover the original key and plaintext, given only the resulting ciphertext. Nowadays, encryption schemes are considered weak if they are vulnerable to attacks much less devastating and much less feasible than this.

There are a lot of open and interesting questions in the field of cryptography, but surprisingly, we already have the answer to one of the more natural questions to ask: "Is there an encryption scheme that cannot be broken?". It turns out that yes, there is. This encryption scheme is called the **one-time pad**. The one-time pad works as follows:

```
Consider a plaintext $P$, and suppose Alice wants to send $P$ to Bob.
Let $K$ be a randomly generated key of the same length $P$,
possessed by both Alice and Bob.
Alice computes $C = P \oplus$
$K$ - i.e. the bitwise xor of the message and the key. This is the ciphertext.
Alice sends $C$ to Bob.
Bob computes $C \oplus$
$K$. Due to the properties of the xor function, $C \oplus K =$
$(P \oplus K) \oplus K = P \oplus (K \oplus K) = P \oplus (0 \dots 0) =$
$P$, and Bob has recovered the original plaintext $P$.
```

As mentioned above, cryptanalysis of this scheme is impossible in a very strong sense. The proof is by an information-theoretic argument due to Shannon (1949), and essentially it involves an argument showing that if an eavesdropper Eve is listening in to the communication between Alice and Bob, when she sees the ciphertext C, she gains exactly zero additional information about the original plaintext P. The way the key is picked ensures that every feasible plaintext might have produced every feasible ciphertext with exactly the same probability ($\frac{1}{2^{len(K)}}$), so knowing C does not make any plaintexts seem any more or less probable than they were in the first place. This property of the One Time Pad scheme is called "perfect security".

As you might have suspected, there is a catch. The catch is that the strength of the One Time Pad's encryption does not come from the (trivially simple) encryption algorithm, but from the key length. To communicate 20MB of encrypted information, Alice and Bob would have to pre-emptively agree on a 20MB secret key. Before sending any ciphertext, they would first have to somehow communicate this key to each other - and then, after this ordeal, they would have to communicate a new key all over again every time they would want to send each other new information.

The key, of course, must remain secret, and Alice and Bob are no more likely to successfully pull off this transfer than they are to just secretly send each other the plaintext they wanted to send in the first place. Due to these considerations, for most applications, the One Time Pad is considered infeasible.

So how might we go about 'fixing' the One Time Pad? Let's look at two possible approaches: The hard way and the easy way.

3.1 The Hard Way: Pseudorandom Generators (PRG)

Perhaps if Alice can't generate huge amounts of random bits and then communicate the resulting key, she can use a generator which will generate the bits for her. This way, Alice and Bob won't have to communicate the whole key in advance - the generator will generate the bits both for her and for Bob

The first obvious problem here is that if Eve gets her hands on this generator, she will also have the bitstream necessary to decrypt all communication between Alice and Bob. In that scenario all communication between Alice and Bob, and whomever else may be using the device, is broken, retroactively and forever. One should take into account here the admonition known as Kerkhoffs' Principle - a cryptosystem's security should depend only on the key (or, as formulated by Shannon, "The enemy knows the system"). When evaluating a cryptosystem's security, our first assumption should be that adversaries will have access to the encryption machinery.

Enter a contraption called a **pseudorandom generator (PRG)**. A pseudorandom generator accepts as input a key K and produces, in a completely deterministic fashion, a bitstream that to the outside observer seems random. If a PRG works as advertised, Alice and Bob can have their cake and eat it too: They covertly exchange a relatively short key K and from then on rely on the bitstream generated by the PRG from K. To decrypt their communication, Eve would, in principle, have to get her hands on K (which ought to be periodically changed, to mitigate the disclosure of information in such an eventuality).

It has been proven, in rigorous terms, that using a true pseudorandom generator, it is possible to build an encryption scheme exactly as secure as the One Time Pad without the latter's crippling drawback of the infeasible key length. The only problem is that no one has so far managed to build a true pseudorandom generator that we know of, and no one even knows if such a beast actually exists. Still, the concept is compelling enough that many ciphers operate on the same principle, using not-quite-perfect-PRGs and hoping that in practice they turn out to be resilient enough, and their imprefections do not leave them open to attack. These are called **stream ciphers**.

By far the most well-known stream cipher is RC4, designed 1987 by Ron Rivest. While deprecated due to a number of weaknesses, it is used in a variety of applications, including Skype, SSH, TLS and the PDF format. The important point about stream ciphers for our purposes is that given a key k, a stream cipher produces a keystream K which is xored with the plaintext P to obtain the ciphertext, similarly to how a One Time Pad is used.

3.2 The Easy Way: Reusing the Key

Suppose both Alice and Bob possess their, let's say, 20MB of One Time Pad key that they have managed to exchange. Suppose they run out of random bits to use as the key. Instead of going through the trouble of sending each other a new key, as we have posited above, perhaps we ought to consider how it would do for them to just use their old key again.

This is certainly the most intuitive solution to the above problem. Indeed, there is historical precedent of resorting to it: When war broke out between Nazi Germany and the Soviet Union in June 1941, demand for encrypted communication within the Soviet ranks spiked, and Soviet code generators began reusing pages of randomly-generated One Time Pads to keep up with demand. Unfortunately, paraphrasing 20th-century American Journalist H. L. Mencken:

For every complex problem there is a solution that is simple, neat and wrong.

The One Time Pad is unbreakable; the One Time Pad with the same keystream having been reused twice is broken. By this we don't mean "broken" in the common modern sense, where one might say "break" and mean a theoretical attack which would require enough computation time for the universe to naturally shrivel and die in the meantime; It is broken in the sense that if you hide two ciphertext strings encrypted with the same key somewhere in 6Kb of random garbage, your laptop is powerful enough to find the two strings and break the encryption while you take a five-minute coffee break.

How so? Well, suppose Alice has two plaintexts p_1, p_2, and further suppose she encrypts each of them using the same keystream k and obtains the corresponding ciphertexts c_1, c_2. By the definition of how a One Time Pad works, we have $c_1 = p_1 \oplus k$ and $c_2 = p_2 \oplus k$. Now, Alice sends c_1, c_2 which are both visible to Eve, who is listening in to the connection. Eve is therefore able to compute $c_1 \oplus c_2$. But due to the properties of the XOR operation, we have:

$$c_1 \oplus c_2 = (p_1 \oplus k) \oplus (p_2 \oplus k) = (p_1 \oplus p_2) \oplus (k \oplus k) = (p_1 \oplus p_2) \oplus 0 = p_1 \oplus p_2$$

Eve can compute the xor of the two original plaintexts. It doesn't even matter what the key k is. It is important to note that this holds true not only in the case where Alice used a classic One Time Pad as described above, but also in the case where Alice used a PRG which was fed the same random seed for both encryptions, and restarted right before each encryption started. That's because given the same seed and starting from the same state, the PRG produces the same keystream both times.

This is, in and of itself, very alarming. In a robust encryption scheme we don't expect an eavesdropper to be able to directly extract information about the plaintext, much less in such a total and direct fashion. But - as mentioned above - it is not merely a red flag; it can be used to 'fish' out encrypted strings from other data without any prior knowledge of whether they even exist, and from there one may proceed to break the encryption, in some cases with alarming ease (see figure 1). The focus of our effort will be the first phase - We will show that not only can we find the offsets and lengths of the ciphertexts

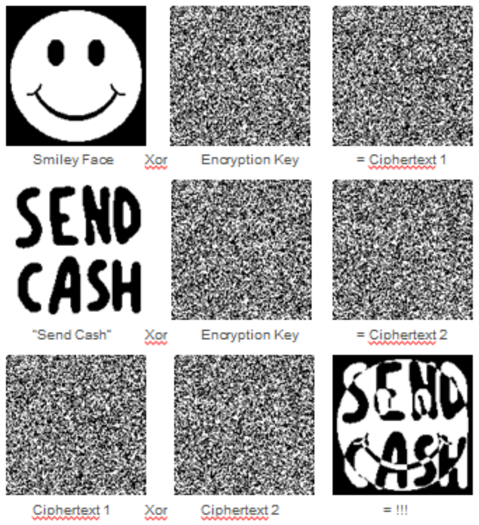


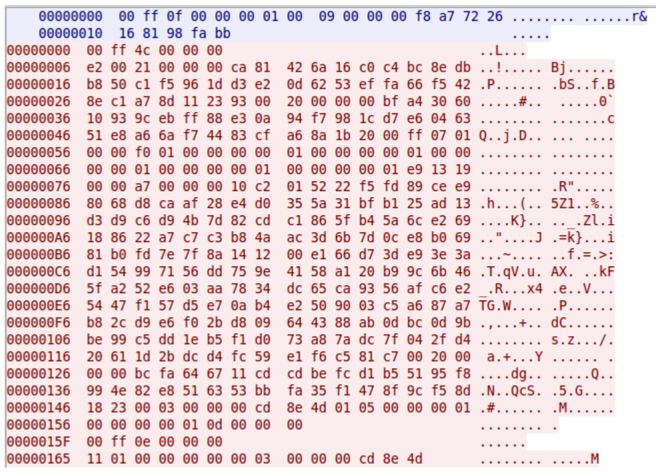
Figure 1. Visual Demonstration of vulnerability in the two-time-pad

in a file that we know in advance to contain this vulnerability - the "needle in a hay stack" problem of fishing out the encrypted strings. We will show how it is possible to go into a text 'blind' without having any prior knowledge of the contents, detect key-reuse ciphertexts and produce their lengths and offsets, all automatically with a quadratic-time heuristic algorithm. Step 2 is a completely different problem, with some significant advances already present in the literature (see for example Mason, Watkins, Eisner and Stubblefield (2006) and their approach based on Smoothed n-gram Language Models), and is beyond the scope of this text.

4 The Two-Time Pad in The Wild

Recently a specimen of the **Ramnit** malware came to our attention. Ramnit is a worm, detected first in 2010 and altered in 2011 to steal FTP credentials, browser cookie information and online banking credentials from victims. The sample produced outbound communication via port 443, which is a standard port for SSL communication, but the protocol being used was clearly not SSL but rather some kind of proprietary protocol (see figure 2).

A reverse-engineering effort revealed that among other features, the protocol supported encrypted objects. Those were marked explicitly by a special 'data type' field preceding the object, and several of them could be sent over during one transaction, or indeed even within a single piece of the TCP conversation. Each object of this type was encrypted using RC4, with the PRG being reset for every encryption, and with the same hard-coded key used every time (the key in this sample was the word 'black').



```
    00000000  00 ff 0f 00 00 00 01 00  09 00 00 00 f8 a7 72 26 ........ ......r&
    00000010  16 81 98 fa bb                                   .....
00000000  00 ff 4c 00 00 00                                ..L...
00000006  e2 00 21 00 00 00 ca 81  42 6a 16 c0 c4 bc 8e db ..!..... Bj......
00000016  b8 50 c1 f5 96 1d d3 e2  0d 62 53 ef fa 66 f5 42 .P...... .bS..f.B
00000026  8e c1 a7 8d 11 23 93 00  20 00 00 00 bf a4 30 60 .....#..  .....0`
00000036  10 93 9c eb ff 88 e3 0a  94 f7 98 1c d7 e6 04 63 ........ .......c
00000046  51 e8 a6 6a f7 44 83 cf  a6 8a 1b 20 00 ff 07 01 Q..j.D.. ... ....
00000056  00 00 f0 01 00 00 00 00  01 00 00 00 00 01 00 00 ........ ........
00000066  00 00 01 00 00 00 00 01  00 00 00 00 01 e9 13 19 ........ ........
00000076  00 00 a7 00 00 00 10 c2  01 52 22 f5 fd 89 ce e9 ........ .R".....
00000086  80 68 d8 ca af 28 e4 d0  35 5a 31 bf b1 25 ad 13 .h...(.. 5Z1..%..
00000096  d3 d9 c6 d9 4b 7d 82 cd  c1 86 5f b4 5a 6c e2 69 ....K}.. .._.Zl.i
000000A6  18 86 22 a7 c7 c3 b8 4a  ac 3d 6b 7d 0c e8 b0 69 .."....J .=k}...i
000000B6  81 b0 fd 7e 7f 8a 14 12  00 e1 66 d7 3d e9 3e 3a ...~.... ..f.=.>:
000000C6  d1 54 99 71 56 dd 75 9e  41 58 a1 20 b9 9c 6b 46 .T.qV.u. AX. ..kF
000000D6  5f a2 52 e6 03 aa 78 34  dc 65 ca 93 56 af c6 e2 _.R...x4 .e..V...
000000E6  54 47 f1 57 d5 e7 0a b4  e2 50 90 03 c5 a6 87 a7 TG.W.... .P......
000000F6  b8 2c d9 e6 f0 2b d8 09  64 43 88 ab 0d bc 0d 9b .,...+.. dC......
00000106  be 99 c5 dd 1e b5 f1 d0  73 a8 7a dc 7f 04 2f d4 ........ s.z.../.
00000116  20 61 1d 2b dc d4 fc 59  e1 f6 c5 81 c7 00 20 00  a.+...Y ...... .
00000126  00 00 bc fa 64 67 11 cd  cd be fc d1 b5 51 95 f8 ....dg.. .....Q..
00000136  99 4e 82 e8 51 63 53 bb  fa 35 f1 47 8f 9c f5 8d .N..QcS. .5.G....
00000146  18 23 00 03 00 00 00 cd  8e 4d 01 05 00 00 00 01 .#...... .M......
00000156  00 00 00 00 01 0d 00 00  00                      ........ .
0000015F  00 ff 0e 00 00 00                                ......
00000165  11 01 00 00 00 00 00 03  00 00 00 cd 8e 4d       ........ .....M
```

Figure 2. Ramnit communication, as recorded by Wireshark

Due to the PRG being reset after every encryption, this scenario exactly fits the requirements that we have outlined earlier - the same keystream is used twice to encrypt different plaintexts in different parts of the text. Based on the theory we have outlined above, we ought to be able to mount an attack on this cryptographic vulnerability.

5 Motivation

If we manage to mount such an attack, we will (hopefully) gain the ability to look at traffic without any prior information and identify that 1. a stream cipher has been used and 2. the same key has been reused to encrypt different plaintexts. This is a fruitful goal for several reasons:

1. Detecting encryption is not at all a trivial problem. Text with high entropy is a starting point, but beyond that we hit a large stumbling block. For many ciphers, there is no known algorithm to tell apart keystream bits from random noise. It just so happens that there **is** such an algorithm for RC4 (see Mantin & Shamir, 2001), but it requires access to many, random, different keystreams, and we have much less to go on - a single batch of ciphertext, the location of which we don't know and the existence of which we're not even sure of.

2. By detecting stream cipher key reuse, we put a finger on a very specific behavior that is a red flag in nearly every possible situation. Sophomoric crypto implementations are a point of interest not only in malicious traffic, but even if found in legitimate data. Combined with other malware IOCs (traffic, static and dynamic), this behavior can form damning evidence of a process.

6 The Algorithm

Our goal is to find same-key ciphertexts in a text $T = t_1 t_2 \dots t_n$ which may or may not contain ciphertexts at all. The main intuition behind our game plan is the visualisation of the space $T^{\oplus} = (t_i \oplus t_j \mid i, j \leq n)$. Suppose for example we have a data string with ciphertext and garbage in it, as follows:

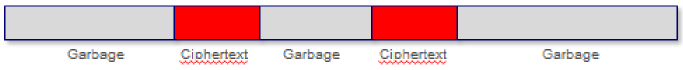


Figure 3. Visualisation of a possible string of data, T

We can visualize the space $T^{\oplus}$ as follows:

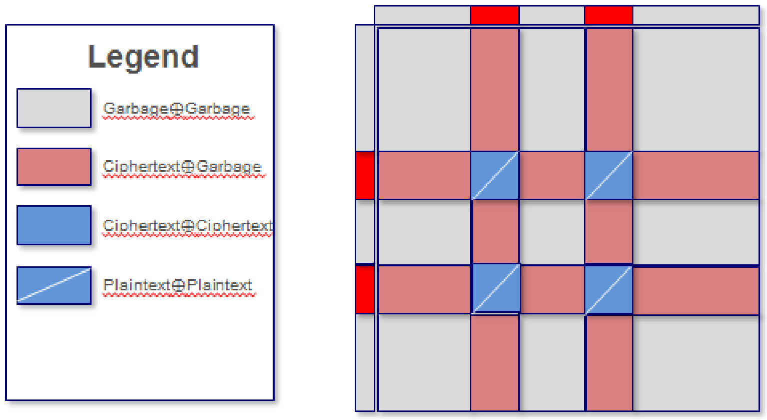


Figure 4. Visualisation of the space $T^{\oplus}$

In particular, what interests us is the places where two different ciphertexts are perfectly aligned - such that, reading diagonally, we can actually see $c_1 \oplus c_2 = p_1 \oplus p_2$ (the white lines in figure 4). The reason we are interested in this part of the space is that plaintext xored with plaintext has a distinctive distribution - not quite as distinctive as plaintext, but hopefully distinctive enough for our purposes. What we have left to do is to devise an algorithm that can fish out those white lines out from $T^{\oplus}$. (Note that the chart is symmetrical along the main diagonal, and the main diagonal is 0 since it consists of data xored with itself; so in this case, the real point of interest is the diagonal on the lower right).

Our pivot is the distinctive distribution along the white diagonals. Scanning along the chart diagonals each in turn, we treat each byte as evidence either for or against the hypothesis "we are currently on a plaintext $\oplus$ plaintext string"; if we see a string of bytes that contains enough evidence, we add it to the result set. It, in turn, maps to two different offsets in the original text T which contain two different ciphertext that were originally different plaintext, encrypted by the same keystream. Note that as for the longest ciphertext which appears in T, recovering the whole of it is inherently impossible using this method, because there are not enough bytes of other ciphertexts to xor against.

Of course, the above was mere hand-waving; the sticking point is the formalism - how we choose to define "evidence", what we mean by "enough evidence" and how well the resulting algorithm can expect to operate in practice. Our first urgent order of business is formalizing "evidence", which is done by applying a basic result from probability theory called **Bayes' Theorem**. The theorem, due to 18th-century statistician Thomas Bayes, is a tool that relates probabilistic reasoning to our "evidence for a hypothesis" scenario. It states:

$$P(H|E) = P(H)\frac{P(E|H)}{P(E)} \tag{1}$$

Literally: The probability of a hypothesis H being true in light of new evidence E equals the probability of H before E was known (called the **prior probability**), multiplied by a scaling factor which is the ratio between two probabilities: the numerator is the probability of E if we assume H is true (the **conditional probability**) and the denominator is the probability of E being true if we do not assume anything about H.

Before using the theorem as-is we will introduce two impotant modifications. By the theorem we have:

$$p(E|\neg H) = P(E)\frac{p(\neg H|E)}{P(\neg H)} \tag{2}$$

Substitute the value of $P(E) = \frac{p(E|\neg H)p(\neg H)}{p(\neg H|E)}$ into the original form of the theorem and obtain:

$$\frac{p(H|E)}{p(\neg H|E)} = \frac{p(H)}{p(\neg H)}\frac{p(E|H)}{p(E|\neg H)} \tag{3}$$

We thus obtain a version of the theorem which can be used for the **odds** of H (defined as the term $\frac{p(H)}{p(\neg H)}$) which, in our case, will be easier to calculate. Further, for an independent set of events $E_1, E_2 \ldots E_n$, we have:

$$\frac{p(H|E_1, E_2, \ldots E_n)}{p(\neg H|E_1, E_2, \ldots E_n)} = \frac{p(H)}{p(\neg H)}\frac{p(E_1, E_2, \ldots E_n|H)}{p(E_1, E_2, \ldots E_n|\neg H)} = \frac{p(H)}{p(\neg H)}\prod_{i=1}^{n}\frac{p(E_i|H)}{p(E_i|\neg H)} \tag{4}$$

With the first equality due to 4 and the second due to the independence of the events E_i. Finally, we apply a logarithm to both sides of the equation to obtain a log-odds form:

$$\log\left(\frac{p(H|E_1, E_2, \ldots E_n)}{p(\neg H|E_1, E_2, \ldots E_n)}\right) - \log\left(\frac{p(H)}{p(\neg H)}\right) = \sum_{i=1}^{n}\log\left(\frac{p(E_i|H)}{p(E_i|\neg H)}\right) \tag{5}$$

This last equation has an interesting interpration; the difference on the left-hand side can be construed as the difference in belief states in H, while the right-hand side can be construed as the sum of available evidence, with the term $\log\left(\frac{p(E_i|H)}{p(E_i|\neg H)}\right)$ representing the "amount of evidenc contributed by E to the hypothesis H". Note that this may be negative, and in fact may attain any value between $-\infty$ and ∞. As evidence for a proposition H mounts, the value $\log\left(\frac{p(H|E_1,E_2,\ldots E_n)}{p(\neg H|E_1,E_2,\ldots E_n)}\right)$ grows. In that scenario, in the probability space induced by assuming the truth of $E_1, E_2, \ldots E_n$, the ratio $\frac{p(H)}{p(\neg H)}$ grows and the (uniquely determined) corresponding value of $p(H)$ grows towards 1.0 accordingly.

5 gives a rigorous formulation for our intuition that "the difference in our belief state in H before and after the evidence is equal to the sum of the evidence strength".

Armed with this formulation, we can now return to the original problem. We will set our hypothesis H to be "the xor-values we are currently looking at are aligned ciphertext$\oplus$ciphertext bytes, properly aligned such that they are equivalent to plaintext$\oplus$plaintext bytes". We will scan the space $T^{\oplus}$diagonally, treating every individual byte as a piece of evidence with value

$$E_{byte} = \log\left(\frac{p(\text{two xored plaintext bytes output this byte})}{p(\text{something which isn't two xored plaintexts output this byte})}\right)$$

These value in the numerator requires the byte distribution of plaintext; it should ideally depend on prior knowledge regarding the kind of plaintext we are looking for, but a more generic distribution will do. The important thing is that is should encode a reasonable amount of information about how some bytes are more likely to appear in plaintext, and others are not. The algorithm proceeds according to the following pseudocode:

```
"good vibes" = 0
"bad vibes" = 0
For each diagonal in the xorspace:
    For each byte in the diagonal:
        if it has positive log-odds:
            reset bad vibes
            add log-odds to "good vibes"
        else:
            add log-odds to "bad vibes"
    If this is the last byte, or we have too many "bad vibes":
        end this streak, start a new streak
        If we collected enough "good vibes" in the streak we just discarded:
            add that streak to our result set
```

As mentioned before, one of our goals is to formulate a precise definition of what we mean by "enough" good vibes and "too many" bad vibes. The approach we chose is computing an estimate for what probability of a false positive would be tolerable for us.

First we should note that our strictness should depend on the length of the text we are observing. In a very long text, we can expect coincidences to happen, some of which would have appeared significant in isolation. In a text of size N, we will be looking at $\binom{n}{2} = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$ bytes, and at most this number of streaks. A natural "normalization" offers itself at this point: Set our expected FP ratio for a streak to $\alpha = \frac{2}{N^2}$ to offset this factor. This is equivalent to setting an evidence threshold of $\log\left(\frac{1}{\alpha}\right) = 2\log(N) - 1$ bits of evidence (Note: Since we are measuring log-odds, instead of α we should technically use $\beta = \frac{2}{N^2-2}$, but it doesn't change our asymptotic approximation. When measuring evidence as outlined earlier and using a base-2 logarithm, it is parlance to say that the evidence is measured in bits).

To get a grasp of the results we can expect, we can look at our assumed plaintext distribution and explicitly compute the characteristics of log-odd bits obtained from the average genuine xortext⊕xortext byte. Suppose it is distributed with mean μ and standard deviation of σ; then the "typical" number of characters exactly long enough to be picked up by the algorithm in a text of length N is

$$L(N) = \left\lceil \frac{2\log(N) - 1}{\mu} \right\rceil$$

This formula should give us an idea of how many characters should be "enough", but it doesn't give us any estimate on the algorithm's probability of success (the old joke comes to mind about the statistician who drowned in a swimming pool which was, on average, 20cm deep). In those kinds of cases - where we are OK in the average case, but we want to get a lower bound for the overall chance that we're OK - we may resort to a form of **Chebyshev's Inequality:**

$$p(|\overline{X} - \mu| \geq k\sigma) \leq \frac{1}{nk^2}$$

Chebyshev's inequality bounds from above the chance that a random variable will deviate "too much" from the mean. Specifically, what this version says is that if we look at the mean of n i.i.d. (independent and identically distributed) variables $X_1.X_2, \ldots X_n$, the probability that it will deviate k standard deviations from μ (the mean of each of the X_i) is at most $\frac{1}{nk^2}$.

Suppose we have a text of length $M \geq L(N)$ and treat the evidence contributions of the bytes as i.i.d. variables $X_1, X_2, \ldots X_M$. By definition, in that case we have $\mu \geq \frac{2\log(N)-1}{M}$. For detection of this text to succeeed, we want the mean evidence contribution of our bytes to be at least $\frac{2\log(N)-1}{M}$. In particular, this means that if the mean byte contribution deviates from μ by less than $\left|\frac{2\log(N)-1}{M} - \mu\right|$, detection will succeed. This is illustrated in figure 5.

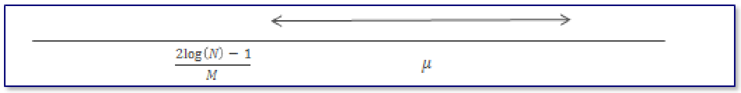


Figure 5. Space of possible mean contributions of text bytes.
The two-sided arrow spans the values of x where $|x-\mu| \le \left|\frac{2\log(N)-1}{M} - \mu\right|$.
Note that all such values are greater than $\frac{2\log(N)-1}{M}$.

We therefore obtain, by the above and by Chebyshev's inequality:

$$p_{fail} \le p\left(|\overline{X} - \mu| \ge \left|\frac{2\log(N)-1}{M} - \mu\right|\right) \le \frac{\sigma^2}{M\left|\frac{2\log(N)-1}{M} - \mu\right|^2}$$

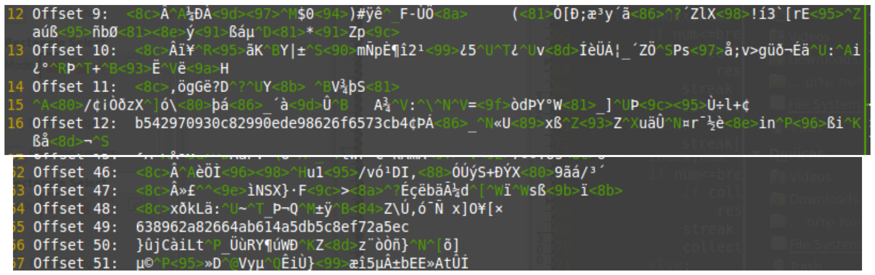
Note the factor of M appearing twice - as we obtain more bytes contributing evidence, the amount of evidence we expect grows and the probability that the evidence accumulated will deviate too much from the amount we expect shrinks; both are to our advantage.

One last modification we must make is to address the concern that ciphertext$\oplus$ciphertext is not the only thing that has the same distribution as plaintext$\oplus$plaintext; another thing with the exact same distribution is **actual** plaintext$\oplus$plaintext. To avoid false positives of this nature, when computing the term $p(E|\neg H)$ we chose $\neg H$ to be either "byte generated by xor of noise" and "the two relevant bytes are plaintext", so as to gain a pessimistic estimate of $\frac{p(H|E)}{p(\neg H|E)}$; false positives are much more of an issue in this setting than false negatives).

7 Proof of Concept - The Ramnit Case

In the case of Ramnit traffic, we have $N = 81$ (the length of the entire data packet transmitted), with two ciphertexts of length 32 embedded in the traffic. We chose a plaintext distribution that looks for hexadecimal hashes (hexadigits uniform distribution) and computed its parameters: $\mu \approx 1.839$ and $\sigma = 1.424$. By those parameters, we have "typical" sufficient length for detection $L(N) = \left\lceil\frac{2\log(N)-1}{\mu}\right\rceil = 7$, and a $p(fail)$ upper bound of $\frac{\sigma^2}{M\left|\frac{2\log(N)-1}{M} - \mu\right|^2} = 3\%$ (meaning, we expect to succeed with probability at least 97%).

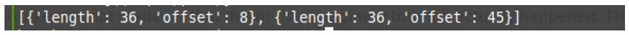
The ciphertexts in the Ramnit traffic are at offsets 12 and 49. The plaintexts can be seen in figure 6 (brute-forced, for the sake of comparison):



```
Offset 9:  <8c>Â^A¼ÐÀ<9d><97>^M$0<94>)#ÿê^_F-ÙÖ<8a>      (<81>Ó[Ð;æ³y´ã<86>^?´ZlX<98>!í3`[rE<95>^Z
aúß<95>ñbØ<81><8e>ý<91>ßáµ^D<81>*<91>Zp<9c>
Offset 10:  <8c>Âï¥^R<95>ãK^BY|±^S<90>mÑpÈ¶î2¹<99>¿5^U^T¿^Uv<8d>ÍèÜÁ¦_´ZÖ^SPs<97>å;v>güð¬Éä^U:^Ai
¿°^RÞ^T+^B<93>Ë^Vë<9a>H
Offset 11:  <8c>,ögGë?D^?^UY<8b> ^BV¾þS<81>
^A<80>/¢iÒðzX^]ó\<80>þá<86>_´à<9d>Û^B   A¾^V:^\^N^V=<9f>òdÞY°W<81>_]^UÞ<9c><95>Ù÷l+¢
Offset 12:  b542970930c82990ede98626f6573cb4¢ÞÁ<86>_^N«U<89>xß^Z<93>Z^XuäÛ^N¤r¯½è<8e>in^P<96>ßi^K
ßå<8d>¬^S
Offset 46:  <8c>Â^AèÖÌ<96><98>^Hu1<95>/vó¹DI,<88>ÓÚýS+ÐÝX<80>9ãá/³´
Offset 47:  <8c>Â»£^^<9e>ìNSX}·F<9c>><8a>^?ÉçëbäĀ¼d^[^Wï^Wsß<9b>ï<8b>
Offset 48:  <8c>xðkLä:^U~^T_Þ¬Q^M±ÿ^B<84>Z\Ú,ó¯Ñ x]O¥[×
Offset 49:  638962a82664ab614a5db5c8ef72a5ec
Offset 50:  }ûjCàiLt^P_ÜùRY¶úWÐ^KZ<8d>z¨òÒñ}^N^[õ]
Offset 51:  µ©^P<95>»D^@Vyµ^QÊiÙ}<99>æî5µÂ±bEE»AtÛÍ
```

Figure 6. Plaintexts and their location, revealed by obtaining the key and running a brute force search

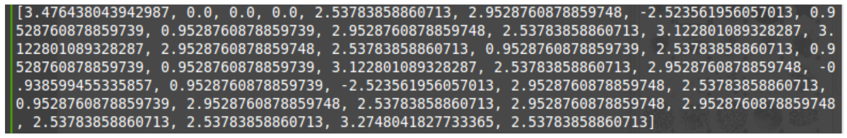
Running the algorithm on the Ramnit traffic reports the two ciphertexts - almost. An interesting quirk is the reporting of 4 leading extra bytes that do not actually belong to any of the ciphertexts, as seen in figure 7.



```
[{'length': 36, 'offset': 8}, {'length': 36, 'offset': 45}]
```

Figure 7. Output of the algirthm. 4 "extra" initial bytes are reported.

We decided to present this anomaly as is, instead of "patching" it, for illustrative purposes. To understand the reason the algorithm believed that those four extra bytes were also originally plaintext, we turn to the array of bits of evidence contributed by every byte in the vicinity:



```
[3.476438043942987, 0.0, 0.0, 0.0, 2.53783858860713, 2.9528760878859748, -2.523561956057013, 0.9
528760878859739, 0.9528760878859739, 2.9528760878859748, 2.53783858860713, 3.122801089328287, 3.
122801089328287, 2.9528760878859748, 2.53783858860713, 0.9528760878859739, 2.53783858860713, 0.9
528760878859739, 0.9528760878859739, 3.122801089328287, 2.53783858860713, 2.9528760878859748, -0
.938599455335857, 0.9528760878859739, -2.523561956057013, 2.9528760878859748, 2.53783858860713,
0.9528760878859739, 2.9528760878859748, 2.53783858860713, 2.9528760878859748, 2.9528760878859748
, 2.53783858860713, 2.53783858860713, 3.2748041827733365, 2.53783858860713]
```

Figure 7. Evidence contributed in the xorspace, starting from coordinates (8,45).

Note the 3 xor-bytes with 0.0 log-odds. This means that for those bytes, it was exactly as likely for them to be generated by xoring plaintext with plaintext as it was to be generated by xoring noise with noise. Intuitively, this is a very outstanding event for one byte, never mind 3 in a row. Looking at the actual string at that offset, we find the culprits - Null Bytes. Under pretty loose conditions, they will always be exactly as probable when xoring ciphertext and xoring plaintext, and therefore the adjustment we made to account for plaintext in the input (choosing the more pessimistic $\neg H$) allows for null bytes to "hitchhike" and be considered a continuation of a true positive. Since they cannot cause false positives in and of themselves (due to their contribution of 0 evidence towards H), and only attach themselves to already existing positives, this is more of a peculiarity than an issue.

8 Conclusion

The approach we have outlined has several advantages. It allows us to go into a text completely blind, without any prior information on our input, and heuristically determine whether a Vernam-style cipher has been used with they key reused at least twice - including the actual offsets and lengths of the ciphertext. It was conceived to exploit a cryptographic weakness encountered in traffic, but can just as easily be applied to any other situation where we suspect the same weakness might arise.

Points of interest and future research include: showing that the bound above behaves nicely even for a more “generic” plaintext distribution; improving the bound parameters by considering byte n-grams instead of single bytes; improving the algorithm’s running time, or proving the impossibility of such; giving a rigorous analysis for an upper bound for the probability of false positives (as was done with false negatives); and integrating an implementation of this approach and of the known approaches of actually breaking the encryption once the ciphertexts have been detected.