Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS

Christina Garman, *Johns Hopkins University;* Kenneth G. Paterson and Thyla Van der Merwe, *University of London*

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/garman

This paper is included in the Proceedings of the 24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-939133-11-3

Attacks Only Get Better: Password Recovery Attacks Against RC4 in TLS

Christina Garman
Johns Hopkins University
cgarman@cs.jhu.edu

Kenneth G. Paterson
Royal Holloway, University of London
kenny.paterson@rhul.ac.uk

Thyla van der Merwe
Royal Holloway, University of London
thyla.vandermerwe.2012@rhul.ac.uk

Abstract

Despite recent high-profile attacks on the RC4 algorithm in TLS, its usage is still running at about 30% of all TLS traffic. We provide new attacks against RC4 in TLS that are focussed on recovering user passwords, still the pre-eminent means of user authentication on the Internet today. Our new attacks use a generally applicable Bayesian inference approach to transform *a priori* information about passwords in combination with gathered ciphertexts into *a posteriori* likelihoods for passwords. We report on extensive simulations of the attacks. We also report on a "proof of concept" implementation of the attacks for a specific application layer protocol, namely BasicAuth. Our work validates the truism that attacks only get better with time: we obtain good success rates in recovering user passwords with 2^{26} encryptions, whereas the previous generation of attacks required around 2^{34} encryptions to recover an HTTP session cookie.

1 Introduction

TLS in all current versions allows RC4 to be used as its bulk encryption mechanism. Attacks on RC4 in TLS were first presented in 2013 in [2] (see also [13, 16]). Since then, usage of RC4 in TLS has declined, but it still accounted for around 30% of all TLS connections in March 2015.[1] Moreover, the majority of websites still support RC4[2] and a small proportion of websites *only* support RC4.[3]

[1]According to data obtained from the International Computer Science Institute (ICSI) Certificate Notary project, which collects statistics from live upstream SSL/TLS traffic in a passive manner; see `http://notary.icsi.berkeley.edu`.

[2]According to statistics obtained from SSL Pulse; see `https://www.trustworthyinternet.org/ssl-pulse/`.

[3]Amounting to 0.79% according to a January 2015 survey of about 400,000 of the Alexa top 1 million sites; see `https://securitypitfalls.wordpress.com/2015/02/01/january-2015-scan-results/`.

We describe attacks recovering TLS-protected passwords whose ciphertext requirements are significantly reduced compared to those of [2]. Instead of the 2^{34} ciphertexts that were needed for recovering 16-byte, base64-encoded secure cookies in [2], our attacks now require around 2^{26} ciphertexts. We also describe a proof-of-concept implementation of these attacks against a specific application-layer protocol making use of passwords, namely BasicAuth.

1.1 Our Contributions

We obtain our improved attacks by revisiting the statistical methods of [2], refining, extending and applying them to the specific problem of recovering TLS-protected passwords. Passwords are a good target for our attacks because they are still very widely used on the Internet for providing user authentication in protocols like BasicAuth and IMAP, with TLS being used to prevent them being passively eavesdropped. To build effective attacks, we need to find and exploit systems in which users' passwords are automatically and repeatedly sent under the protection of TLS, so that sufficiently many ciphertexts can be gathered for our statistical analyses.

Bayesian analysis We present a formal Bayesian analysis that combines an *a priori* plaintext distribution with keystream distribution statistics to produce *a posteriori* plaintext likelihoods. This analysis formalises and extends the procedure followed in [2] for single-byte attacks. There, only keystream distribution statistics were used (specifically, biases in the individual bytes in the early portion of the RC4 keystream) and plaintexts were assumed to be uniformly distributed, while here we also exploit (partial) knowledge of the plaintext distribution to produce a more accurate estimate of the *a posteriori* likelihoods. This yields a procedure that is optimal (in the sense of yielding a maximum *a posteriori* estimate for the plaintext) if the plaintext distribution is known exactly.

In the context of password recovery, an *estimate* for the *a priori* plaintext distribution can be empirically formed by using data from password breaches or by synthetically constructing password dictionaries. We will demonstrate, via simulations, that this Bayesian approach improves performance (measured in terms of success rate of plaintext recovery for a given number of ciphertexts) compared to the approach in [2].

Our Bayesian analysis concerns vectors of consecutive plaintext bytes, which is appropriate given passwords as the plaintext target. This, however, means that the keystream distribution statistics also need to be for vectors of consecutive keystream bytes. Such statistics do not exist in the prior literature on RC4, except for the Fluher-McGrew biases [10] (which supply the distributions for adjacent byte pairs far down the keystream). Fortunately, in the early bytes of the RC4 keystream, the single-byte biases are dominant enough that a simple product distribution can be used as a reasonable estimate for the distribution on vectors of keystream bytes. We also show how to build a more accurate approximation to the relevant keystream distributions using double-byte distributions. (Obtaining the double-byte distributions to a suitable degree of accuracy consumed roughly 4800 core-days of computation; for details see the full version [12].) This approximation is not only more accurate but also *necessary* when the target plaintext is located further down the stream, where the single-byte biases disappear and where double-byte biases become dominant. Indeed, our double-byte-based approximation to the keystream distribution on vectors can be used to smoothly interpolate between the region where single-byte biases dominate and where the double-byte biases come into play (which is exhibited as a fairly sharp transition around position 256 in the keystream).

In the end, what we obtain is a formal algorithm that estimates the likelihood of each password in a dictionary based on both the *a priori* password distribution and the observed ciphertexts. This formal algorithm is amenable to efficient implementation using either the single-byte based product distribution for keystreams or the double-byte-based approximation to the distribution on keystreams. The dominant terms in the running time for both of the resulting algorithms is $\mathcal{O}(nN)$ where n is the length of the target password and N is the size of the dictionary used in the attack.

An advantage of our new algorithms over the previous work in [2] is that they output a value for the likelihood of each password candidate, enabling these to be ranked and then tried in order of descending likelihood.

Note that our Bayesian approach is quite general and not limited to recovery of passwords, nor to RC4 – it can be applied whenever the plaintext distribution is approximately known, where the same plaintext is repeatedly encrypted, and where the stream cipher used for encryption has known biases in either single bytes or adjacent pairs of bytes.

Evaluation We evaluate and compare our password recovery algorithms through extensive simulations, exploring the relationships between the main parameters of our attack:

- The length n of the target password.
- The number S of available encryptions of the password.
- The starting position r of the password in the plaintext stream.
- The size N of the dictionary used in the attack, and the availability (or not) of an *a priori* password distribution for this dictionary.
- The number of attempts T made (meaning that our algorithm is considered successful if it ranks the correct password amongst the top T passwords, i.e. the T passwords with highest likelihoods as computed by the algorithm).
- Which of our two algorithms is used (the one computing the keystream statistics using the product distribution or the one using a double-byte-based approximation).
- Whether the passwords are Base64 encoded before being transmitted, or are sent as raw ASCII/Unicode.

Given the many possible parameter settings and the cost of performing simulations, we focus on comparing the performance with all but one or two parameters or variables being fixed in each instance.

Proofs of concept Our final contribution is to apply our attacks to specific and widely-deployed applications making use of passwords over TLS: BasicAuth and (in the full version [12]), IMAP. We introduce BasicAuth and describe a proof-of-concept implementation of our attacks against it, giving an indication of the practicality of our attacks. We do the same for IMAP in the full version [12].

For both applications, we have significant success rates with only $S = 2^{26}$ ciphertexts, in contrast to the roughly 2^{34} ciphertexts required in [2]. This is because we are able to force the target passwords into the first 256 bytes of plaintext, where the large single-byte biases in RC4 keystreams come into play. For example, with $S = 2^{26}$ ciphertexts, we would expect to recover a length 6 BasicAuth password with 44.5% success rate after $T = 5$ attempts; the rate rises to 64.4% if $T = 100$ attempts are made. In practice, many sites do not configure any limit on the number of BasicAuth attempts made by a client; moreover a study [5] showed that 84% of websites surveyed allowed for up to 100 password guesses (though these sites were not necessarily using BasicAuth as their authentication mechanism). As we will show, our result compares very favourably to the previous attacks and to random guessing of passwords without any reference to the ciphertexts.

However, there is a downside too: to make use of the early, single-byte biases in RC4 keystreams, we have to repeatedly cause TLS connections to be closed and new ones to be opened. Because of latency in the TLS Handshake Protocol, this leads to a significant slowdown in the wall clock running time of the attack; for $S = 2^{26}$, a latency of 100ms, and exploiting browsers' propensity to open multiple parallel connections, we estimate a running time of around 300 hours for the attack. This is still more than 6 times faster than the 2000 hours estimated in [2]. Furthermore, the attack's running time reduces proportionately to the latency of the TLS Handshake Protocol, so in environments where the client and server are close – for example in a LAN – the execution time could be a few tens of hours.

2 Further Background

2.1 The RC4 algorithm

Originally a proprietary stream cipher designed by Ron Rivest in 1987, RC4 is remarkably fast when implemented in software and has a very simple description. Details of the cipher were leaked in 1994 and the cipher has been subject to public analysis and study ever since.

RC4 allows for variable-length key sizes, anywhere from 40 to 256 bits, and consists of two algorithms, namely, a *key scheduling algorithm* (KSA) and a *pseudo-random generation algorithm* (PRGA). The KSA takes as input an l-byte key and produces the initial internal state $st_0 = (i, j, \mathscr{S})$ for the PRGA; $\mathscr{S}$ is the canonical representation of a permutation of the numbers from 0 to 255 where the permutation is a function of the l-byte key, and i and j are indices for $\mathscr{S}$. The KSA is specified in Algorithm 1 where K represents the l-byte key array and $\mathscr{S}$ the 256-byte state array. Given the internal state st_r, the PRGA will generate a keystream byte Z_{r+1} as specified in Algorithm 2.

Algorithm 1: RC4 key scheduling (KSA)

```
input  : key $K$ of $l$ bytes
output : initial internal state $st_0$
begin
    for $i = 0$ to 255 do
        $\mathscr{S}[i] \leftarrow i$
    $j \leftarrow 0$
    for $i = 0$ to 255 do
        $j \leftarrow j + \mathscr{S}[i] + K[i \bmod l]$
        swap($\mathscr{S}[i], \mathscr{S}[j]$)
    $i, j \leftarrow 0$
    $st_0 \leftarrow (i, j, \mathscr{S})$
    return $st_0$
```

Algorithm 2: RC4 keystream generator (PRGA)

```
input  : internal state $st_r$
output : keystream byte $Z_{r+1}$
         updated internal state $st_{r+1}$
begin
    parse $(i, j, \mathscr{S}) \leftarrow st_r$
    $i \leftarrow i + 1$
    $j \leftarrow j + \mathscr{S}[i]$
    swap($\mathscr{S}[i], \mathscr{S}[j]$)
    $Z_{r+1} \leftarrow \mathscr{S}[\mathscr{S}[i] + \mathscr{S}[j]]$
    $st_{r+1} \leftarrow (i, j, \mathscr{S})$
    return $(Z_{r+1}, st_{r+1})$
```

2.2 Single-byte biases in the RC4 Keystream

RC4 has several cryptographic weaknesses, notably the existence of various biases in the RC4 keystream, see for example [2, 10, 14, 15, 19]. Large single-byte biases are prominent in the early postions of the RC4 keystream. Mantin and Shamir [15] observed the first of these biases, in Z_2 (the second byte of the RC4 keystream), and showed how to exploit it in what they called a *broadcast attack*, wherein the same plaintext is repeatedly encrypted under different keys. AlFardan *et al.* [2] performed large-scale computations to estimate these early biases, using 2^{45} keystreams to compute the single-byte keystream distributions in the first 256 output positions. They also provided a statistical approach to recovering plaintext bytes in the broadcast attack scenario, and explored its exploitation in TLS. Much of the new bias behaviour they observed was subsequently explained in [18]. Unfortunately, from an attacker's perspective, the single-byte biases die away very quickly beyond position 256 in the RC4 keystream. This means that they can only be used in attacks to extract plaintext bytes which are found close to the start of plaintext streams. This was a significant complicating factor in the attacks of [2], where, because of the behaviour of HTTP in modern browsers, the target HTTP secure cookies were not so located.

2.3 Double-byte biases in the RC4 Keystream

Fluhrer and McGrew [10] showed that there are biases in adjacent bytes in RC4 keystreams, and that these so-called double-byte biases are persistent throughout the keystream. The presence of these long-term biases (and the absence of any other similarly-sized double-byte biases) was confirmed computationally in [2]. AlFardan *et al.* [2] also exploited these biases in their double-byte attack to recover HTTP secure cookies.

Because we wish to exploit double-byte biases in early portions of the RC4 keystream and because the analysis of [10] assumes the RC4 permutation $\mathscr{S}$ is uniformly random (which is not the case for early keystream bytes), we carried out extensive computations to estimate the initial double-byte keystream distributions: we used roughly 4800 core-days of computation to generate 2^{44} RC4 keystreams for random 128-bit RC4 keys (as used in TLS); we used these keystreams to estimate the double-byte keystream distributions for RC4 in the first 512 positions.

While the gross behaviour that we observed is dominated by products of the known single-byte biases in the first 256 positions and by the Fluhrer-McGrew biases in the later positions, we did observe some new and interesting double-byte biases. Since these are likely to be of independent interest to researchers working on RC4, we report in more detail on this aspect of our work in the full version [12].

2.4 RC4 and the TLS Record Protocol

We provide an overview of the TLS Record Protocol with RC4 selected as the method for encryption and direct the reader to [2, 6, 7, 8] for further details.

Application data to be protected by TLS, i.e, a sequence of bytes or a record R, is processed as follows: An 8-byte sequence number `SQN`, a 5-byte header `HDR` and R are concatenated to form the input to an HMAC function. We let T denote the resulting output of this function. In the case of RC4 encryption, the plaintext, $P = T||R$, is XORed byte-per-byte with the RC4 keystream. In other words,

$$C_r = P_r \oplus Z_r,$$

for the r^{th} bytes of the ciphertext, plaintext and RC4 keystream respectively (for $r = 1, 2, 3 \ldots$). The data that is transmitted has the form `HDR`$||C$, where C is the concatenation of the individual ciphertext bytes.

The RC4 algorithm is intialized in the standard way at the start of each TLS connection with a 128-bit encryption key. This key, K, is derived from the TLS master secret that is established during the TLS Handshake Protocol; K is either established via the the full TLS Handshake Protocol or TLS session resumption. The first few bytes to be protected by RC4 encryption is a `Finished` message of the TLS Handshake Protocol. We do not target this record in our attacks since this message is not constant over multiple sessions. The exact size of this message is important in dictating how far down the keystream our target plaintext will be located; in turn this determines whether or not it can be recovered using only single-byte biases. A common size is 36 bytes, but the exact size depends on the output size of the TLS PRF used in computing the `Finished` message and of the hash function used in the HMAC algorithm in the record protocol.

Decryption is the reverse of the process described above. As noted in [2], any error in decryption is treated as fatal – an error message is sent to the sender and all cryptographic material, including the RC4 key, is disposed of. This enables an active attacker to force the use of new encryption and MAC keys: the attacker can induce session termination, followed by a new session being established when the next message is sent over TLS, by simply modifying a TLS Record Protocol message. This could be used to ensure that the target plaintext in an attack is repeatedly sent under the protection of a fresh RC4 key. However, this approach is relatively expensive since it involves a rerun of the full TLS Handshake Protocol, involving multiple public key operations and, more importantly, the latency involved in an exchange of 4 messages (2 complete round-trips) on the wire. A better approach is to cause the TCP connection carrying the TLS traffic to close, either by injecting sequences of `FIN` and `ACK` messages in both directions, or by injecting a `RST` message in both directions. This causes the TLS *connection* to be terminated, but not the TLS *session* (assuming the session is marked as "resumable" which is typically the case). This behaviour is codified in [8, Section 7.2.1]. Now when the next message is sent over TLS, a TLS session resumption instance of the Handshake Protocol is executed to establish a fresh key for RC4. This avoids the expensive public key operations and reduces the TLS latency to 1 round-trip before application data can be sent. On large sites, session resumption is usually handled by making use of TLS session tickets [17] on the server-side.

2.5 Passwords

Text-based passwords are arguably the dominant mechanism for authenticating users to web-based services and computer systems. As is to be expected of user-selected secrets, passwords do not follow uniform distributions. Various password breaches of recent years, including the Adobe breach of 150 million records in 2013 and the RockYou leak of 32.6 million passwords in 2009, attest to this with passwords such as `123456` and `password` frequently being counted amongst the most popular.[4] For example, our own analysis of the RockYou password data set confirmed this: the number of unique passwords in the RockYou dataset is 14,344,391, meaning that (on average) each password was repeated 2.2 times, and we indeed found the most common password to be `123456` (accounting for about 0.9% of the entire data set). Our later simulations will make extensive use of the RockYou data set as an attack dictionary. A more-fine grained analysis of it can be found in [20]. We also make use of data from the Singles.org breach for generating our target passwords. Singles.org is a now-defunct Christian dating website that was breached in 2009; religiously-inspired passwords such as `jesus` and `angel` appear with high frequency in its 12,234 distinct entries, making its frequency distribution quite different from that of the RockYou set.

There is extensive literature regarding the reasons for poor password selection and usage, including [1, 9, 21, 22]. In [4], Bonneau formalised a number of different metrics for analysing password distributions and studied a corpus of 70M Yahoo! passwords (collected in a privacy-preserving manner). His work highlights the importance of careful validation of password guessing attacks, in particular, the problem of estimating attack complexities in the face of passwords that occur rarely – perhaps uniquely – in a data set, the so-called *hapax legomena* problem. The approach to validation that we adopt benefits from the analysis of [4], as explained further in Section 4.

3 Plaintext Recovery via Bayesian Analysis

In this section, we present a formal Bayesian analysis of plaintext recovery attacks in the broadcast setting for stream ciphers. We then apply this to the problem of extracting passwords, specialising the formal analysis and making it implementable in practice based only on the single-byte and double-byte keystream distributions.

3.1 Formal Bayesian Analysis

Suppose we have a candidate set of N plaintexts, denoted $\mathscr{X}$, with the *a priori* probability of an element $x \in \mathscr{X}$ being denoted p_x. We assume for simplicity that all the candidates consist of byte strings of the same length n. For example $\mathscr{X}$ might consist of all the passwords of a given length n from some breach data set, and then p_x can be computed as the relative frequency of x in the data set. If the frequency data is not available, then the uniform distribution on $\mathscr{X}$ can be assumed.

[4]A comprehensive list of data breaches, including password breaches, can be found at `http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/`.

Next, suppose that a plaintext from $\mathscr{X}$ is encrypted S times, each time under independent, random keys using a stream cipher such as RC4. Suppose also that the first character of the plaintext always occurs in the same position r in the plaintext stream in each encryption. Let $c = (c_{ij})$ denote the $S \times n$ matrix of bytes in which row i, denoted $c^{(i)}$ for $0 \leq i < S$, is a vector of n bytes corresponding to the values in positions $r, \ldots, r+n-1$ in ciphertext i. Let X be the random variable denoting the (unknown) value of the plaintext.

We wish to form a maximum a posteriori (MAP) estimate for X, given the observed data c and the *a priori* probability distribution p_x, that is, we wish to maximise $\Pr(X = x \mid C = c)$ where C is a random variable corresponding to the matrix of ciphertext bytes.

Using Bayes' theorem, we have

$$\Pr(X = x \mid C = c) = \Pr(C = c \mid X = x) \cdot \frac{\Pr(X = x)}{\Pr(C = c)}.$$

Here the term $\Pr(X = x)$ corresponds to the *a priori* distribution p_x on $\mathscr{X}$. The term $\Pr(C = c)$ is independent of the choice of x (as can be seen by writing $\Pr(C = c) = \sum_{x \in \mathscr{X}} \Pr(C = c \mid X = x) \cdot \Pr(X = x)$). Since we are only interested in maximising $\Pr(X = x \mid C = c)$, we ignore this term henceforth.

Now, since ciphertexts are formed by XORing keystreams z and plaintext x, we can write

$$\Pr(C = c \mid X = x) = \Pr(W = w)$$

where w is the $S \times n$ matrix formed by XORing each row of c with the vector x and W is a corresponding random variable. Then to maximise $\Pr(X = x \mid C = c)$, it suffices to maximise the value of

$$\Pr(X = x) \cdot \Pr(W = w)$$

over $x \in \mathscr{X}$. Let $w^{(i)}$ denote the i-th row of the matrix w, so $w^{(i)} = c^{(i)} \oplus x$. Then $w^{(i)}$ can be thought of as a vector of keystream bytes (coming from positions $r, \ldots, r+n-1$) induced by the candidate x, and we can write

$$\Pr(W = w) = \prod_{i=0}^{S-1} \Pr(Z = w^{(i)})$$

where, on the right-hand side of the above equation, Z denotes a random variable corresponding to a vector of bytes of length n starting from position r in the keystream. Writing $\mathscr{B} = \{\texttt{0x00}, \ldots, \texttt{0xFF}\}$ for the set of bytes, we can rewrite this as:

$$\Pr(W = w) = \prod_{z \in \mathscr{B}^n} \Pr(Z = z)^{N_{x,z}}$$

where the product is taken over all possible byte strings of length n and $N_{x,z}$ is defined as:

$$N_{x,z} = |\{i : z = c^{(i)} \oplus x\}_{0 \leq i < S}|,$$

that is, $N_{x,z}$ counts the number of occurrences of vector z in the rows of the matrix formed by XORing each row of c with candidate x. Putting everything together, our objective is to compute for each candidate $x \in \mathscr{X}$ the value:

$$\Pr(X = x) \cdot \prod_{z \in \mathscr{B}^n} \Pr(Z = z)^{N_{x,z}}$$

and then to rank these values in order to determine the most likely candidate(s).

Notice that the expressions here involve terms $\Pr(Z = z)$ which are probabilities of occurrence for n consecutive bytes of keystream. Such estimates are not generally available in the literature, and for the values of n we are interested in (corresponding to putative password lengths), obtaining accurate estimates for them by sampling many keystreams would be computationally prohibitive. Moreover, the product $\prod_{z \in \mathscr{B}^n}$ involves 2^{8n} terms and is not amenable to calculation. Thus we must turn to approximate methods to make further progress.

Note also that taking $n = 1$ in the above analysis, we obtain exactly the same approach as was used in the single-byte attack in [2], except that we include the *a priori* probabilities $\Pr(X = x)$ whereas these were (implicitly) assumed to be uniform in [2].

3.2 Using a Product Distribution

Our task is to derive simplified ways of computing the expression

$$\Pr(X = x) \cdot \prod_{z \in \mathscr{B}^n} \Pr(Z = z)^{N_{x,z}}$$

and then apply these to produce efficient algorithms for computing (approximate) likelihoods of candidates $x \in \mathscr{X}$.

The simplest approach is to assume that the n bytes of the keystreams can be treated independently. For RC4, this is actually a very good approximation in the regime where single-byte biases dominate (that is, in the first 256 positions). Thus, writing $Z = (Z_r, \ldots, Z_{r+n-1})$ and $z = (z_r, \ldots, z_{r+n-1})$ (with the subscript r denoting the position of the first keystream byte of interest), we have:

$$\Pr(Z = z) \approx \prod_{j=0}^{n-1} \Pr(Z_{r+j} = z_{r+j}) = \prod_{j=0}^{n-1} p_{r+j,z}$$

where now the probabilities appearing on the right-hand side are single-byte keystream probabilities, as reported in [2] for example. Then writing $x = (x_0, \ldots, x_{n-1})$ and rearranging terms, we obtain:

$$\prod_{z \in \mathscr{B}^n} \Pr(Z = z)^{N_{x,z}} \approx \prod_{j=0}^{n-1} \prod_{z \in \mathscr{B}} p_{r+j,z}^{N_{x_j,z,j}}$$

where $N_{y,z,j} = |\{i : z = c_{i,j} \oplus y\}_{0 \le i < S}|$ counts (now for single bytes instead of length n vectors of bytes) the number of occurrences of byte z in the column vector formed by XORing column j of c with a candidate byte y.

Notice that, as in [2], the counters $N_{y,z,j}$ for $y \in \mathscr{B}$ can all be computed efficiently by permuting the counters $N_{\mathtt{0x00},z,j}$, these being simply counters for the number of occurrences of each byte value z in column j of the ciphertext matrix c.

In practice, it is more convenient to work with logarithms, converting products into sums, so that we evaluate for each candidate $x = (x_0, \ldots, x_{n-1})$ an expression of the form

$$\gamma_x := \log(p_x) + \sum_{j=0}^{n-1} \sum_{z \in \mathscr{B}} N_{x_j,z,j} \log(p_{r+j,z}).$$

Given a large set of candidates $\mathscr{X}$, we can streamline the computation by first computing the counters $N_{y,z,j}$, then, for each possible byte value y, the value of the inner sum $\sum_{z \in \mathscr{B}} N_{y,z,j} \log(p_{r+j,z})$, and then reusing these individual values across all the relevant candidates x for which $x_j = y$. This reduces the evaluation of γ_x for a single candidate x to $n + 1$ additions of real numbers.

The above procedure, including the various optimizations, is specified as an attack in Algorithm 3. We refer to it as our single-byte attack because of its reliance on the single-byte keystream probabilities $p_{r+j,z}$. It outputs a collection of approximate log likelihoods $\{\gamma_x : x \in \mathscr{X}\}$ for each candidate $x \in \mathscr{X}$. These can be further processed to extract, for example, the candidate with the highest score, or the top T candidates.

3.3 Double-byte-based Approximation

We continue to write $Z = (Z_r, \ldots, Z_{r+n-1})$ and $z = (z_r, \ldots, z_{r+n-1})$ and aim to find an approximation for $\Pr(Z = z)$ which lends itself to efficient computation of approximate log likelihoods as in our first algorithm. Now we rely on the double-byte keystream distribution, writing

$$p_{s,z_1,z_2} := \Pr((Z_s, Z_{s+1}) = (z_1, z_2)), \quad s \ge 1, k_1, k_2 \in \mathscr{B}$$

for the probabilities of observing bytes (z_1, z_2) in the RC4 keystream in positions $(s, s+1)$. We estimated these probabilities for r in the range $1 \le r \le 511$ using 2^{44} RC4 keystreams – for details, see the full version; for larger r, these are well approximated by the Fluhrer-McGrew biases [10] (as was verified in [2]).

We now make the Markovian assumption that, for each j,

$$\begin{aligned} &\Pr(Z_j = z_j \mid Z_{j-1} = z_{j-1} \wedge \cdots \wedge Z_0 = z_0) \\ &\qquad \approx \Pr(Z_j = z_j \mid Z_{j-1} = z_{j-1}), \end{aligned}$$

Algorithm 3: Single-byte attack

```
input  : $c_{i,j} : 0 \le i < S, 0 \le j < n$ – array formed from $S$ independent encryptions of fixed $n$-byte candidate $X$
         $r$ – starting position of $X$ in plaintext stream
         $\mathscr{X}$ – collection of $N$ candidates
         $p_x$ – a priori probability of candidates $x \in \mathscr{X}$
         $p_{r+j,z}$ ($0 \le j < n$, $z \in \mathscr{B}$) – single-byte keystream distribution
output : $\{\gamma_x : x \in \mathscr{X}\}$ – set of (approximate) log likelihoods for candidates in $\mathscr{X}$
begin
    for $j = 0$ to $n-1$ do
        for $z =$ 0x00 to 0xFF do
            $N'_{z,j} \leftarrow 0$
    for $j = 0$ to $n-1$ do
        for $i = 0$ to $S-1$ do
            $N'_{c_{i,j},j} \leftarrow N'_{c_{i,j},j} + 1$
    for $j = 0$ to $n-1$ do
        for $y =$ 0x00 to 0xFF do
            for $z =$ 0x00 to 0xFF do
                $N_{y,z,j} \leftarrow N'_{z \oplus y,j}$
            $L_{y,j} = \sum_{z \in \mathscr{B}} N_{y,z,j} \log(p_{r+j,z})$,
    for $x = (x_0, \ldots, x_{n-1}) \in \mathscr{X}$ do
        $\gamma_x \leftarrow \log(p_x) + \sum_{j=0}^{n-1} L_{x_j,j}$
    return $\{\gamma_x : x \in \mathscr{X}\}$
```

meaning that byte j in the keystream can be modelled as depending only on the preceding byte and not on earlier bytes. We can write

$$\Pr(Z_j = z_j \mid Z_{j-1} = z_{j-1}) = \frac{\Pr(Z_j = z_j \wedge Z_{j-1} = z_{j-1})}{\Pr(Z_{j-1} = z_{j-1})}$$

where the numerator can then be replaced by p_{j-1,z_{j-1},z_j} and the denominator by $p_{j-1,z_{j-1}}$, a single-byte keystream probability. Then using an inductive argument and our assumption, we easily obtain:

$$\Pr(Z = z) \approx \frac{\prod_{j=0}^{n-2} p_{r+j,z_j,z_{j+1}}}{\prod_{j=1}^{n-2} p_{r+j,z_j}}$$

giving an approximate expression for our desired probability in terms of single-byte and double-byte probabilities. Notice that if we assume that the adjacent byte pairs are independent, then we have $p_{r+j,z_j,z_{j+1}} = p_{r+j,z_j} \cdot p_{r+j+1,z_{j+1}}$ and the above expression collapses down to the one we derived in the previous subsection.

For candidate x, we again write $x = (x_0, \ldots, x_{n-1})$ and rearranging terms, we obtain:

$$\prod_{z \in \mathscr{B}^n} \Pr(Z = z)^{N_{x,z}} \approx \frac{\prod_{j=0}^{n-2} \prod_{z_1 \in \mathscr{B}} \prod_{z_2 \in \mathscr{B}} p_{r+j,z_1,z_2}^{N_{x_j,x_{j+1},z_1,z_2,j}}}{\prod_{j=1}^{n-2} \prod_{z \in \mathscr{B}} p_{r+j,z}^{N_{x_j,z,r+j}}}$$

where $N_{y_1,y_1,z_1,z_2,j} = |\{i : z_1 = c_{i,j} \oplus y_1 \wedge z_2 = c_{i,j+1} \oplus y_2\}_{0 \le i < S}|$ counts (now for consecutive pairs of bytes) the number of occurrences of bytes (z_1, z_2) in the pair of column vectors formed by XORing columns $(j, j+1)$ of c with candidate bytes (y_1, y_2) (and where $N_{x_j,z,r+j}$ is as in our previous algorithm).

Again, the counters $N_{y_1,y_2,z_1,z_2,j}$ for $y_1, y_2 \in \mathscr{B}$ can all be computed efficiently by permuting the counters $N_{\texttt{0x00},\texttt{0x00},z_1,z_2,j}$, these being simply counters for the number of occurrences of pairs of byte values (z_1, z_2) in column j and $j+1$ of the ciphertext matrix c. As before, we work with logarithms, so that we evaluate for each candidate $x = (x_0, \ldots, x_{n-1})$ an expression of the form

$$\gamma_x := \log(p_x) + \sum_{j=0}^{n-2} \sum_{z_1 \in \mathscr{B}} \sum_{z_2 \in \mathscr{B}} N_{x_j,x_{j+1},z_1,z_2,j} \log(p_{r+j,z_1,z_2}) - \sum_{j=1}^{n-2} \sum_{z \in \mathscr{B}} N_{x_j,z,r+j} \log(p_{r+j,z}).$$

With appropriate pre-computation of the terms $N_{y_1,y_2,z_1,z_2,j} \log(p_{r+j,z_1,z_2})$ and $N_{y,z,r+j} \log(p_{r+j,z})$ for all y_1, y_2 and all y, the computation for each candidate $x \in \mathscr{X}$ can be reduced to roughly $2n$ floating point additions. The pre-computation can be further reduced by computing the terms for only those pairs (y_1, y_2) actually arising in candidates in $\mathscr{X}$ in positions $(j, j+1)$. We use this further optimisation in our implementation.

The above procedure is specified as an attack in Algorithm 4. We refer to it as our double-byte attack because of its reliance on the double-byte keystream probabilities p_{s,z_1,z_2}. It again outputs a collection of approximate log likelihoods $\{\gamma_x : x \in \mathscr{X}\}$ for each candidate $x \in \mathscr{X}$, suitable for further processing. Note that for simplicity of presentation, it involves a quintuply-nested loop to compute the values $N_{y_1,y_2,z_1,z_2,j}$; these values should of course be directly computed from the $(n-1)\cdot 2^{16}$ pre-computed counters $N'_{c_{i,j},c_{i,j+1},j}$ in an in-line fashion using the formula $N_{y_1,y_2,z_1,z_2,j} = N'_{z_1\oplus y_1,z_2\oplus y_2,,j}$.

4 Simulation Results

4.1 Methodology

We performed extensive simulations of both of our attacks, varying the different parameters to evaluate their effects on success rates. We focus on the problem of password recovery, using the RockYou data set as an attack dictionary and the Singles.org data set as the set of target passwords. Except where noted, in each simulation, we performed 256 independent runs of the relevant attack. In each attack in a simulation, we select a password of some fixed length n from the Singles.org password data set according to the known *a priori* probability distribution for that data set, encrypt it S times in different starting positions r using random 128-bit keys for RC4, and then attempt to recover the password from the ciphertexts using the set of all passwords of length n from the entire RockYou data set (14 million passwords) as our candidate set $\mathscr{X}$. We declare success if the target password is found within the top T passwords suggested by the algorithm (according to the approximate likelihood measures γ_x). Our default settings, unless otherwise stated, are $n = 6$ and $T = 5$. Six is the most common password length in the data sets we encountered; $T = 5$ is an arbitrary choice, and we examine the effect of varying T in detail below. We try all values for r between 1 and $256-n+1$, where the single-byte biases dominate the behaviour of the RC4 keystreams. Typical values of S are 2^s where $s \in \{20,22,24,26,28\}$.

Using different data sets for the attack dictionary and the target set from which encrypted passwords are chosen is more realistic than using a single dictionary for both purposes, not least because in a real attack, the exact content and *a priori* distribution of the target set would not be known. This approach also avoids the problem of *hapax legomena* highlighted in [4]. However, this has the effect of limiting the success rates of our attacks to less than 100%, since there are highly likely passwords in the target set (such as `jesus`) that do not occur at all, or only have very low *a priori* probabilities in the attack dictionary, and conversely. Figure 1 compares the use of the

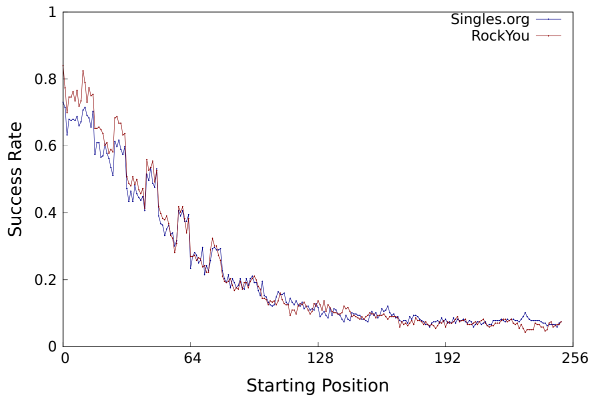


Figure 1: Recovery rate for Singles.org passwords using RockYou data set as dictionary, compared to recovery rate for RockYou passwords using RockYou data set as dictionary ($S = 2^{24}$, $n = 6$, $T = 5$, $1 \le r \le 251$, double-byte attack).

RockYou password distribution to attack Singles.org passwords with the less realistic use of the RockYou password distribution to attack RockYou itself. It can be seen that, for the particular choice of attack parameters ($S = 2^{24}$, $n = 6$, $T = 5$, double-byte attack), the effect on success rate is not particularly large. However, for other attack parameters, as we will see below, we observe a maximum success rate of around 80% for our attacks, whereas we would achieve 100% success rates if we used RockYou against itself. The observed maximum success rate could be increased by augmenting the attack dictionary with synthetically generated, site-specific passwords and by removing RockYou-specific passwords from the attack dictionary. We leave the development and evaluation of these improvements to future work.

Many data sets are available from password breaches. We settled on using RockYou for the attack dictionary because it was one of the biggest data sets in which all passwords and their associated frequencies were available, and because the distribution of passwords, while certainly skewed, was less skewed than for other data sets. We used Singles.org for the target set because the Singles.org breach occurred later than the RockYou breach, so that the former could reasonably used as an attack dictionary for the latter. Moreover, the Singles.org distribution being quite different from that for RockYou makes password recovery against Singles.org using RockYou as a dictionary more challenging for our attacks. A detailed evaluation of the extent to which the success rates of our attacks depend on the choice of attack dictionary and target set is beyond the scope of this current work.

A limitation of our approach as described is that we assume the password length n to be already known. Sev-

Algorithm 4: Double-byte attack

```
input : $c_{i,j} : 0 \leq i < S, 0 \leq j < n$ – array formed from $S$ independent encryptions of fixed $n$-byte candidate $X$
        $r$ – starting position of $X$ in plaintext stream
        $\mathscr{X}$ – collection of $N$ candidates
        $p_x$ – *a priori* probability of candidates $x \in \mathscr{X}$
        $p_{r+j,z}$ $(0 \leq j < n,\ z \in \mathscr{B})$ – single-byte keystream distribution
        $p_{r+j,z_1,z_2}$ $(0 \leq j < n-1,\ z_1, z_2 \in \mathscr{B})$ – double-byte keystream distribution
output: $\{\gamma_x : x \in \mathscr{X}\}$ – set of (approximate) log likelihoods for candidates in $\mathscr{X}$
begin
    for $j = 0$ to $n-2$ do
        for $z_1 = $ 0x00 to 0xFF do
            $N'_{z,j} \leftarrow 0$
            for $z_2 = $ 0x00 to 0xFF do
                $N'_{z_1,z_2,j} \leftarrow 0$
    for $j = 0$ to $n-2$ do
        for $i = 0$ to $S-1$ do
            $N'_{c_{i,j},j} \leftarrow N'_{c_{i,j},j} + 1$
            $N'_{c_{i,j},c_{i,j+1},j} \leftarrow N'_{c_{i,j},c_{i,j+1},j} + 1$
    for $j = 1$ to $n-2$ do
        for $y = $ 0x00 to 0xFF do
            for $z = $ 0x00 to 0xFF do
                $N_{y,z,j} \leftarrow N'_{z \oplus y,j}$
            $L_{y,j} = \sum_{z \in \mathscr{B}} N_{y,z,j} \log(p_{r+j,z})$,
    for $j = 0$ to $n-2$ do
        for $y_1 = $ 0x00 to 0xFF do
            for $y_2 = $ 0x00 to 0xFF do
                for $z_1 = $ 0x00 to 0xFF do
                    for $z_2 = $ 0x00 to 0xFF do
                        $N_{y_1,y_2,z_1,z_2,j} \leftarrow N'_{z_1 \oplus y_1, z_2 \oplus y_2,,j}$
                $L_{y_1,y_2,j} = \sum_{z_1 \in \mathscr{B}} \sum_{z_2 \in \mathscr{B}} N_{y_1,y_2,z_1,z_2,j} \log(p_{r+j,z_1,z_2})$,
    for $x = (x_0, \ldots, x_{n-1}) \in \mathscr{X}$ do
        $\gamma_x \leftarrow \log(p_x) + \sum_{j=0}^{n-2} L_{x_j,x_{j+1},j} - \sum_{j=1}^{n-2} L_{x_j,j}$
    return $\{\gamma_x : x \in \mathscr{X}\}$
```

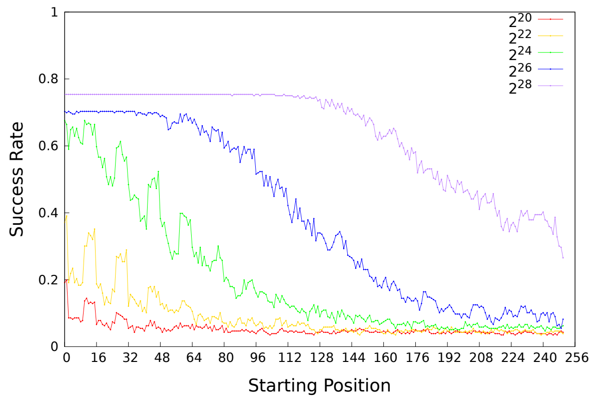


Figure 2: Recovery rates for single-byte algorithm for $S = 2^{20}, \dots, 2^{28}$ ($n = 6$, $T = 5$, $1 \le r \le 251$).

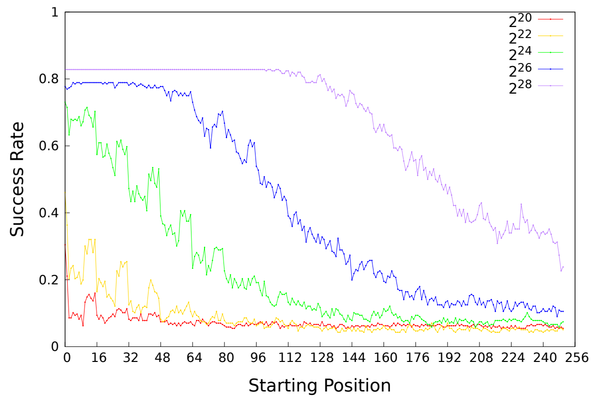


Figure 3: Recovery rates for double-byte algorithm for $S = 2^{20}, \dots, 2^{28}$ ($n = 6$, $T = 5$, $1 \le r \le 251$).

eral solutions to this problem are described in the full version [12].

4.2 Results

Single-Byte Attack We ran the attack described in Algorithm 3 with our default parameters ($n = 6$, $T = 5$, $1 \le r \le 251$) for $S = 2^s$ with $s \in \{20, 22, 24, 26, 28\}$ and evaluated the attack's success rate. We used our default of 256 independent runs per parameter set. The results are shown in Figure 2. We observe that:

- The performance of the attack improves markedly as S, the number of ciphertexts, increases, but the success rate is bounded by 75%. We attribute this to the use of one dictionary (RockYou) to recover passwords from another (Singles.org) – for the same attack parameters, we achieved 100% success rates when using RockYou against RockYou, for example.
- For 2^{24} ciphertexts we see a success rate of greater than 60% for small values of r, the assumed position of the password in the RC4 keystream. We see a drop to below 50% for starting positions greater than 32. We note the effect of the key-length-dependent biases on password recovery; passwords encrypted at starting positions $16\ell - n, 16\ell - n + 1, \dots, 16\ell - 1, 16\ell$, where $\ell = 1, 2, \dots, 6$, have a higher probability of being recovered in comparison to neighbouring starting positions.
- For 2^{28} ciphertexts we observe a success rate of more than 75% for $1 \le r \le 120$.

Double-Byte Attack Analogously, we ran the attack of Algorithm 4 for $S = 2^s$ with $s \in \{20, 22, 24, 26, 28\}$ and our defaults of $n = 6$, $T = 5$. The results for these simulations are shown in Figure 3. Note that:

- Again, at 2^{24} ciphertexts the effect of key-length-dependent biases is visible.
- For 2^{26} ciphertexts we observe a success rate that is greater than 78% for $r \le 48$.

Comparing the Single-Byte Attack with a Naive Algorithm Figure 4 provides a comparison between our single-byte algorithm with $T = 1$ and a naive password recovery attack based on the methods of [2], in which the password bytes are recovered one at a time by selecting the highest likelihood byte value in each position and declaring success if all bytes of the password are recovered correctly. Significant improvement over the naive attack can be observed, particularly for high values of r. For example with $S = 2^{24}$, the naive algorithm essentially has a success rate of zero for every r, whereas our single-byte algorithm has a success rate that exceeds 20% for $1 \le r \le 63$. By way of comparison, an attacker knowing the password length and using the obvious guessing strategy would succeed with probability 4.2% with a single guess, this being the *a priori* probability of the password 123456 amongst all length 6 passwords in the Singles.org dataset (and 123456 being the highest ranked password in the RockYou dictionary, so the first one that an attacker using this strategy with the RockYou dictionary would try). As another example, with $S = 2^{28}$ ciphertexts, a viable recovery rate is observed all the way up to $r = 251$ for our single-byte algorithm, whereas the naive algorithm fails badly beyond $r = 160$ for even this large value of S. Note however that the naive attack can achieve a success rate of 100% for sufficiently large S, whereas our attack cannot. This is because the naive attack directly

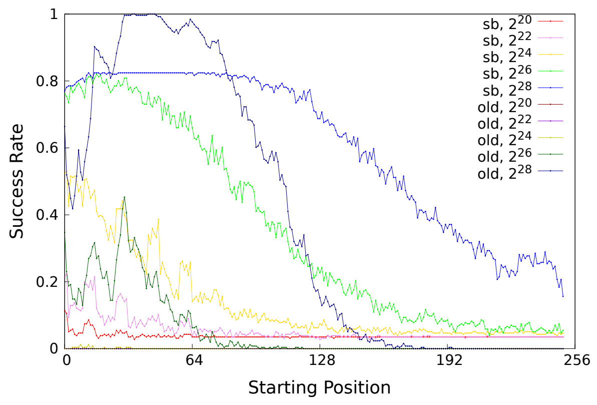


Figure 4: Performance of our single-byte algorithm versus a naive single-byte attack based on the methods of AlFardan *et al.* (labelled “old”) ($n = 6$, $T = 1$, $1 \le r \le 251$).

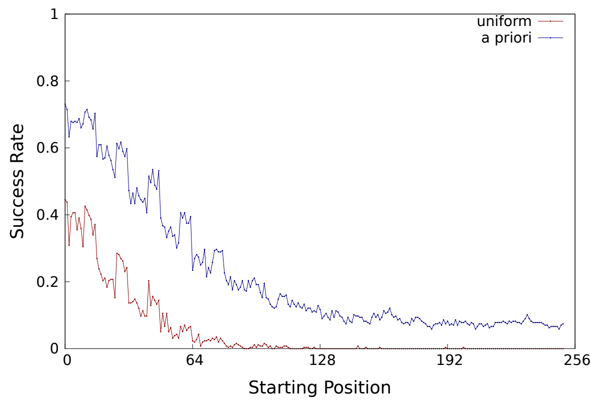


Figure 6: Recovery rate for uniformly distributed passwords versus known *a priori* distribution ($S = 2^{24}$, $n = 6$, $T = 5$, $1 \le r \le 251$, double-byte algorithm).

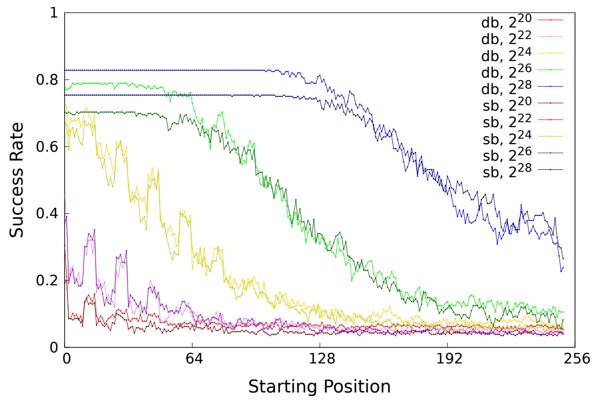


Figure 5: Recovery rate of single-byte versus double-byte algorithm for $S = 2^{20}, \ldots, 2^{28}$ ($n = 6$, $T = 5$, $1 \le r \le 251$).

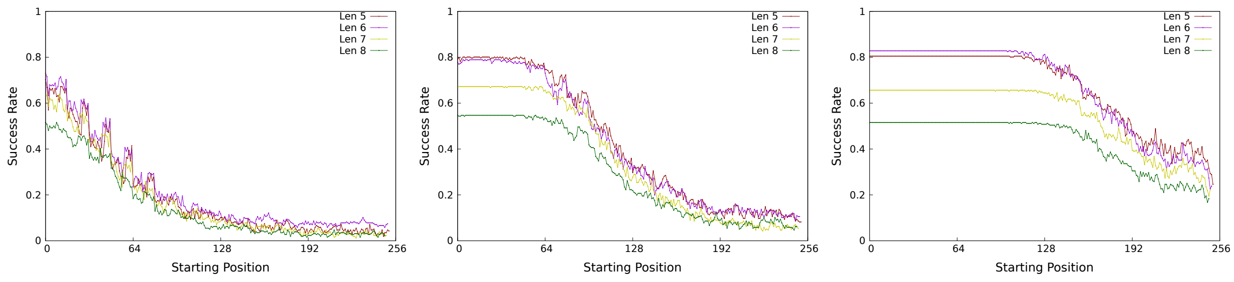
computes a password candidate rather than evaluating the likelihood of candidates from a list which may not contain the target password. On the other hand, our attack trivially supports larger values of T, whereas the naive attack is not so easily modified to enable this feature.

Comparing the Single-Byte and Double-Byte Attacks Figure 5 provides a comparison of our single-byte and double-byte attacks. With all other parameters equal, the success rates are very similar for the initial 256 positions. The reason for this is the absence of many strong double-byte biases that do not arise from the known single-byte biases in the early positions of the RC4 keystream.

Effect of the *a priori* Distribution As a means of testing the extent to which our success rates are influenced by knowledge of the *a priori* probabilities of the candidate passwords, we ran simulations in which we tried to recover passwords sampled correctly from the Singles.org dataset but using a uniform *a priori* distribution for the RockYou-based dictionary used in the attack. Figure 6 shows the results ($S = 2^{24}$, $n = 6$, $T = 5$, double-byte attack) of these simulations, compared to the results we obtain by exploiting the *a priori* probabilities in the attack. It can be seen that a significant gain is made by using the *a priori* probabilities, with the uniform attack's success rate rapidly dropping to zero at around $r = 128$.

Effect of Password Length Figure 7 shows the effect of increasing n, the password length, on recovery rates, with the sub-figures showing the performance of our double-byte attack for different numbers of ciphertexts ($S = 2^s$ with $s \in \{24, 26, 28\}$). Other parameters are set to their default values. As intuition suggests, password recovery becomes more difficult as the length increases. Also notable is that the ceiling on success rate of our attack decreases with increasing n, dropping from more than 80% for $n = 5$ to around 50% for $n = 8$. This is due to the fact that only 48% of the length 8 passwords in the Singles.org data set actually occur in the RockYou attack dictionary: our attack is doing as well as it can in this case, and we would expect stronger performance with an attack dictionary that is better matched to the target site.

Effect of Increasing Try Limit T Recall that the parameter T defines the number of password trials our attacks make. The number of permitted attempts for specific protocols like BasicAuth and IMAP is server-dependent and not mandated in the relevant specifications. Whilst not specific to our chosen protocols, a 2010 study [5] showed that 84% of websites surveyed allowed at least $T = 100$ attempts; many websites appear to actually al-



(a) 2^{24} ciphertexts (b) 2^{26} ciphertexts (c) 2^{28} ciphertexts

Figure 7: Effect of password length on recovery rate ($T = 5$, $1 \leq r \leq 251$, double-byte algorithm).

low $T = \infty$. Figure 8 shows the effect of varying T in our double-byte algorithm for different numbers of ciphertexts ($S = 2^s$ with $s \in \{24, 26, 28\}$). Other parameters are set to their default values. It is clear that allowing large values of T boosts the success rate of the attacks.

Note however that a careful comparison must be made between our attack with parameter T and the success rate of the obvious password guessing attack given T attempts. Such a guessing attack does not require any ciphertexts but instead uses the *a priori* distribution on passwords in the attack dictionary (RockYou) to make guesses for the target password in descending order of probability, the success rate being determined by the *a priori* probabilities of the guessed passwords in the target set (Singles.org). Clearly, our attacks are only of value if they significantly out-perform this ciphertext-less attack.

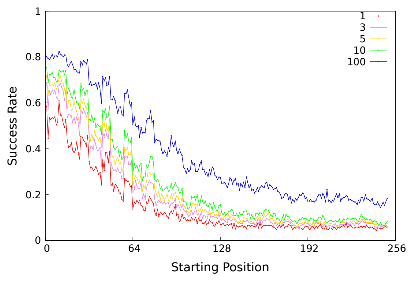
Figure 9 shows the results of plotting $\log_2(T)$ against success rate α for $S = 2^s$ with $s \in \{14, 16, \ldots, 28\}$. The figure then illustrates the value of T necessary in our attack to achieve a given password recovery rate α for different values of S. This measure is related to the α-work-factor metric explored in [4] (though with the added novelty of representing a work factor when one set of passwords is used to recover passwords from a different set). To generate this figure, we used 1024 independent runs rather than the usual 256, but using a fixed set of 1024 passwords sampled according to the *a priori* distribution for Singles.org. This was in an attempt to improve the stability of the results (with small numbers of ciphertexts S, the success rate becomes heavily dependent on the particular set of passwords selected and their *a priori* probabilities, while we wished to have comparability across different values of S).

The success rates shown in Figure 9 are for our double-byte attack with $n = 6$ and $r = 133$, this specific choice of r being motivated by it being the location of passwords for our BasicAuth attack proof-of-concept when the Chrome browser is used (similar results are obtained for other values of r). The graph also shows the corresponding work factor T as a function of α for the guessing attack (labeled "optimal guessing" in the figure).

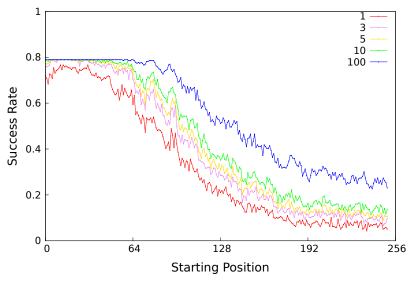
Figure 9 shows that our attack far outperforms the guessing attack for larger values of S, with a significant advantage accruing for $S = 2^{24}$ and above. However, the advantage over the guessing attack for smaller values of S, namely 2^{20} and below, is not significant. This can be attributed to our attack simply not being able to compute stable enough statistics for these small numbers of ciphertexts. In turn, this is because the expected random fluctuations in the keystream distributions overwhelm the small biases; in short, the signal does not sufficiently exceed the noise for these low values of S.

Effect of Base64 Encoding We investigated the effect of Base64 encoding of passwords on recovery rates, since many application layer protocols use such an encoding. The encoding increases the password length, making recovery harder, but also introduces redundancy, potentially helping the recovery process to succeed. Figure 10 shows our simulation results comparing the performance of our double-byte algorithm acting on 6-character passwords and on Base64 encoded versions of them. It is apparent from the figure that the overall effect of the Base64 encoding is to help our attack to succeed. In practice, the start of the target password may not be well-aligned with the Base64 encoding process (for example, part of the last character of the username and/or a delimiter such as ":" may be jointly encoded with part of the first character of the password). This can be handled by building a special-purpose set of candidates $\mathscr{X}$ for each possibility. Handling this requires some care when mounting a real attack against a specific protocol; a detailed analysis is deferred to future work.

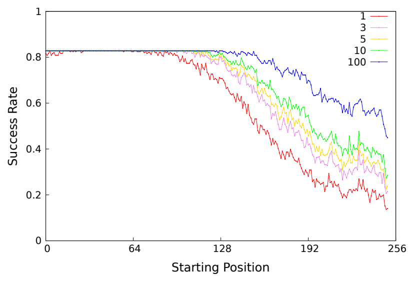
Shifting Attack In certain application protocols and attack environments (such as HTTPS) it is possible for the adversary to incrementally pad the plaintext messages so that the unknown bytes are always aligned with positions having large keystream biases. Our algorithm descriptions



(a) 2^{24} ciphertexts



(b) 2^{26} ciphertexts



(c) 2^{28} ciphertexts

Figure 8: Effect of try limit T on recovery rate ($n = 6$, $1 \leq r \leq 251$, double-byte algorithm).

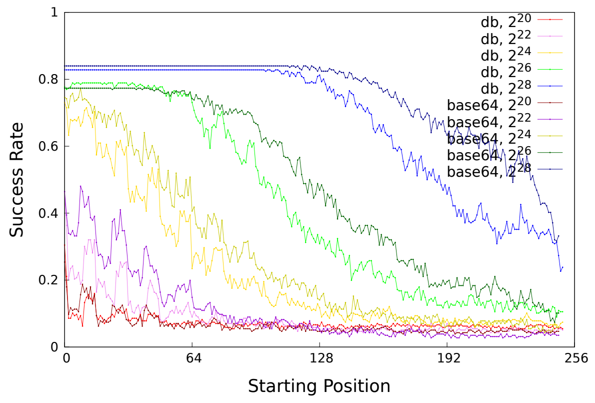


Figure 10: Recovery rate of Base64 encoded password versus a "normal" password for 6-character passwords ($T = 5$, $1 \leq r \leq 251$, double-byte algorithm).

and code are both easily modified to handle this situation, and we have conducted simulations with the resulting shift attack. We report on these simulations in the full version, [12].

5 Practical Validation

In this section we describe proof-of-concept implementations of our attacks against a specific application-layer protocol running over TLS, namely BasicAuth. In the full version [12], we additionally consider the IMAP protocol as a target.

5.1 Introducing BasicAuth

Defined as part of the HTTP/1.0 specification [3] and extended in [11], the Basic Access Authentication scheme (BasicAuth) provides a simple means for controlling access to webpages and other protected resources. In view of its simplicity, the scheme is still very widely used in the enterprise application space. The protocol essentially involves the client sending the server a username and password in Base64 encoded form, and as such, requires the use of a lower-layer secure protocol like TLS to mitigate trivial eavesdropping attacks. Certain web browsers display a login dialog when an initiating challenge message is received from the server and many browsers present users with the option of storing their user credentials in the browser, with the credentials thereafter being automatically presented on behalf of the user.

The client response to the challenge is of the form `Authorization: Basic Base64(userid:password)` where `Base64(·)` denotes the Base64 encoding function (which maps 3 characters at a time onto 4 characters of output).

5.2 Attacking BasicAuth

To obtain a working attack against BasicAuth, we need to ensure that two conditions are met:

- The Base64-encoded password included in the BasicAuth client response can be located sufficiently early in the plaintext stream.
- There is a method for forcing a browser to repeatedly send the BasicAuth client response.

We have observed that the first condition is met for particular browsers, including Google Chrome. For example, we inspected HTTPS traffic sent from Chrome to an iChair server.[5] We observed the user's Base64-encoded password being sent with every HTTP(S) request in the same position in the stream, namely position $r = 133$ (this includes 16 bytes consumed by the client's `Finished` message as well as the 20-bytes consumed by the TLS Record Protocol tag). For Mozilla Firefox, the value of r was the less useful 349.

[5]iChair is a popular system for conference reviewing, widely used in the cryptography research community and available from `http://www.baigneres.net/ichair`. It uses BasicAuth as its user authentication mechanism.

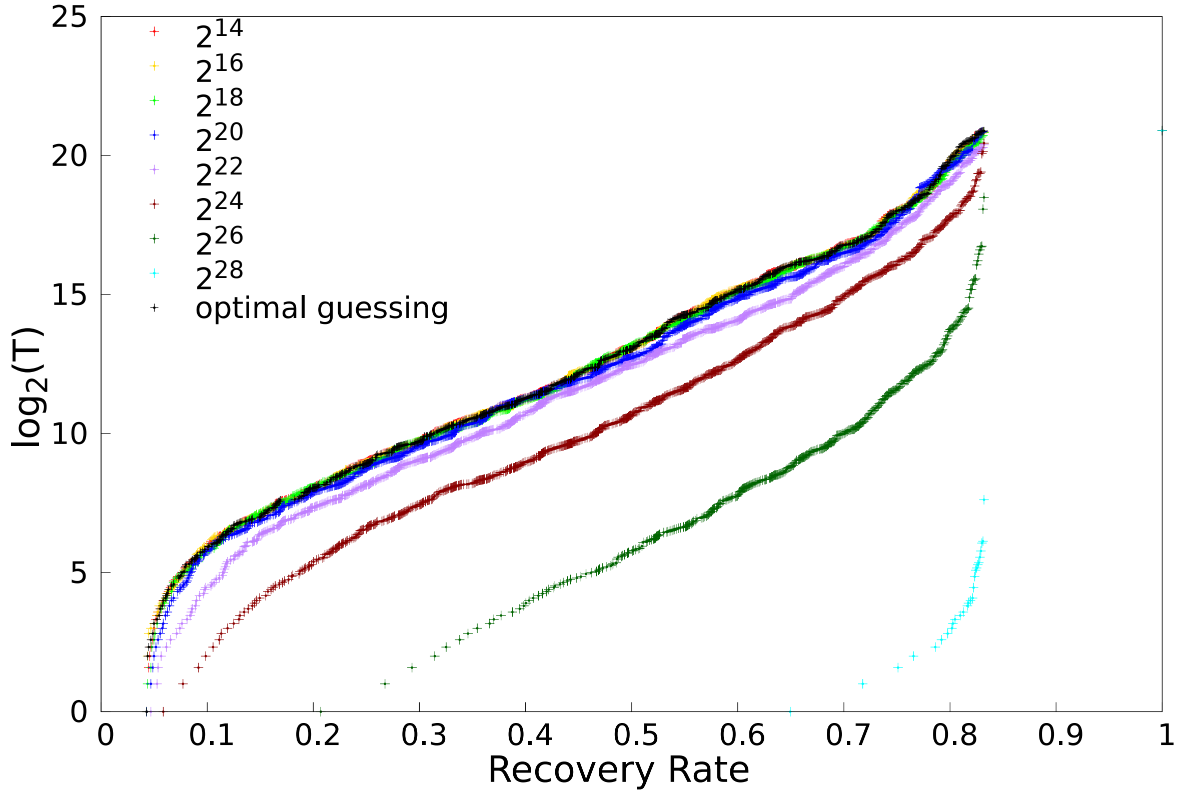


Figure 9: Value of T required to achieve a given password recovery rate α for $S = 2^s$ with $s \in \{14, 16, \ldots, 28\}$ ($n = 6$, $r = 133$, double-byte algorithm).

For the second condition, we adopt the methods used in the BEAST, CRIME and Lucky 13 attacks on TLS, and also used in attacking RC4 in [2]: we assume that the user visits a site `www.evil.com` which loads JavaScript into the user's browser; the JavaScript makes GET or POST requests to the target website at `https://www.good.com` by using `XMLHttpRequest` objects (this is permitted under Cross Origin Resource Sharing (CORS), a mechanism developed to allow JavaScript to make requests to a domain other than the one from which the script originates). The Base64-encoded BasicAuth password is automatically included in each such request. To force the password to be repeatedly encrypted at an early position in the RC4 keystream, we use a MITM attacker to break the TLS connection (by injecting sequences of TCP `FIN` and `ACK` messages into the connection). This requires some careful timing on the part of the JavaScript and the MITM attacker.

We built a proof-of-concept demonstration of these components to illustrate the principles. We set up a virtual network with three virtual machines each running Ubuntu 14.04, kernel version 3.13.0-32. On the first machine, we installed iChair. We configured the iChair web server to use RC4 as its default TLS cipher. The second machine was running the Chrome 38 browser and acted as the client in our attack. We installed the required JavaScript directly on this machine rather than downloading from another site. The third machine acted as the MITM attacker, required to intercept the TLS-protected traffic and to tear-down the TLS connections. We used the Python tool Scapy[6] to run an ARP poisoning attack on the client and server from the MITM so as to be able to intercept packets; with the connection hijacked we were able to force a graceful shutdown of the connection between the client and the server after the password-bearing record had been observed and recorded. We observed that forcing a graceful shutdown of each subsequent connection did allow for TLS resumption (rather than leading to the need for a full TLS Handshake run).

With this setup, the JavaScript running in the client browser sent successive HTTPS GET requests to the

[6]Available at `http://www.secdev.org/projects/scapy/`.

iChair server every 80ms. Our choice of 80ms was motivated by the fact that for our particular configuration, we observed a total time of around 80ms for TLS resumption, delivery of the password-bearing record and the induced shutdown of the TCP connection. This choice enabled us to capture 2^{16} encrypted password-bearing records in 1.6 hours (the somewhat greater than expected time here being due to anomalies in network behaviour). Running at this speed, the attack was stable over a period of hours.

We note that the latency involved in our setup is much lower than would be found in a real network in which the server may be many hops away from the client: between 500ms and 1000ms is typical for establishing an initial TLS connection to a remote site, with the latency being roughly half that for session resumptions. Notably, the cost of public key operations is not the issue, but rather the network latency involved in the round-trips required for TCP connection establishment and then running the TLS Handshake. However, browsers also open up multiple TLS connections in parallel when fetching multiple resources from a site, as a means of reducing the latency perceived by users; the maximum number of concurrent connections per server is 6 for both the Chrome and Firefox browsers (though, we only ever saw roughly half this number in practice, even with low inter-request times). This means that, assuming a TLS resumption latency (including the client's TCP `SYN`, delivery of the password-bearing record and the final, induced TCP `ACK`) of 250ms and the JavaScript is running fast enough to induce the browser to maintain 6 connections in parallel, the amount of time needed to mount an attack with $S = 2^{26}$ would be on the order of 776 hours. If the latency was further reduced to 100ms (because of proximity of the server to the client), the attack execution time would be reduced to 312 hours.

Again setting $n = 6$, $T = 100$, $r = 133$ and using the simulation results displayed in Figure 10, we would expect a success rate of 64.4% for this setup (with $S = 2^{26}$). For $T = 5$, the corresponding success rate would be 44.5%.

We emphasise that we have not executed a complete attack on these scales, but merely demonstrated the feasibility of the attack in our laboratory setup.

6 Conclusion and Open Problems

We have presented plaintext recovery attacks that derive from a formal Bayesian analysis of the problem of estimating plaintext likelihoods given an *a priori* plaintext distribution, suitable keystream distribution information, and a large number of encryptions of a fixed plaintext under independent keys. We applied these ideas to the specific problem of recovering passwords encrypted by the RC4 algorithm with 128-bit keys as used in TLS, though they are of course more generally applicable – to uses of RC4 other than in TLS, and to stream ciphers with non-uniform keystream distributions in general. Using large-scale simulations, we have investigated the performance of these attacks under different settings for the main parameters.

We then studied the applicability of these attacks for a specific application layer protocol, BasicAuth. For certain browsers and clients, the passwords were located at a favourable point in the plaintext stream and we could induce the password to be repeatedly encrypted under fresh, random keys. We built a proof-of-concept implementation of the attack. It was difficult to arrange for the rate of generation of encryptions to be as high as desired for a speedy attack. This was mainly due to the latency associated with TLS connection establishment (even with session resumption) rather than any fundamental barrier.

Good-to-excellent password recovery success rates can be achieved using 2^{24} – 2^{28} ciphertexts in our attacks. We also demonstrated that our single-byte attack for password recovery significantly outperforms a naive password recovery attack based on the ideas of [2]. We observed an improvement over a guessing strategy even for low numbers (2^{22} or 2^{24}) of ciphertexts. By contrast to these numbers, the preferred double-byte attack of [2] required on the order of 2^{34} encryptions to recover a 16-byte cookie, though without incurring the time overheads arising from TLS session resumption that our approach incurs.

Our research has led to the identification of a number of areas for further work:

- Our Bayesian approach can also be applied to the situation where we model the plaintext as a word from a language described as a Markov model with memory. It would be interesting to investigate the extent to which this approach can be applied to either password recovery or more general analysis of, say, typical HTTP traffic.

- We have focussed on the use of the single-byte biases described in [2] and the double-byte biases of Fluhrer and McGrew (and from our own extensive computations for the first 512 keystream positions). Other biases in RC4 keystreams are known, for example [14]. It is a challenge to integrate these in our Bayesian framework, with the aim being to further improve our attacks.

- We identified new double-byte biases early in the RC4 keystream which deserve a theoretical explanation.

- It would be an interesting challenge to develop algorithms for constructing synthetic, site-specific dictionaries along with *a priori* probability distributions. Existing work in this direction includes Marx's WordHound tool.[7]

- We identified several open questions in the discussion of our simulation results, including the effect of the choice of password data sets on success rates, and the evaluation of different methods for recovering the target password's length.

Acknowledgements

We would like to thank Google, Dan Kaminsky at White Ops and Ingo von Maurich at Ruhr Universität Bochum for their generous donation of computing resources. Dan gave us free rein on a 512-core system for the 4800 core-days necessary to perform our double-byte keystream distribution estimates, while resources from Google and Ruhr Universität Bochum were used to conduct our attack simulations. We would also like to thank Alexei Melnikov for acting as our IMAP oracle.

Garman was funded by a generous grant from the Mozilla Foundation and supported by the Office of Naval Research under contract N00014-14-1-0333; Paterson was supported by an EPSRC Leadership Fellowship, EP/H005455/1; van der Merwe was supported by the EPSRC as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London.

References

[1] Adams, A., and Sasse, M. A. Users are not the enemy. *Commun. ACM 42*, 12 (Dec. 1999), 40–46.

[2] AlFardan, N. J., Bernstein, D. J., Paterson, K. G., Poettering, B., and Schuldt, J. C. N. On the Security of RC4 in TLS. In *Proceedings of the 22nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association, pp. 305–320.

[3] Berners-Lee, T., Fielding, R., and Frystyk, H. The Hypertext Transfer Protocol HTTP/1.0. RFC 1945 (Informational), May 1996.

[4] Bonneau, J. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA* (2012), IEEE Computer Society, pp. 538–552.

[5] Bonneau, J., and Preibusch, S. The password thicket: Technical and market failures in human authentication on the web. In *9th Annual Workshop on the Economics of Information Security, WEIS 2010, Harvard University, Cambridge, MA, USA, June 7 - 8* (2010).

[6] Dierks, T., and Allen, C. The TLS Protocol Version 1.0. RFC 2246, Internet Engineering Task Force, Jan. 1999.

[7] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Internet Engineering Task Force, Apr. 2006.

[8] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Internet Engineering Task Force, Aug. 2008.

[7]`https://bitbucket.org/mattinfosec/wordhound`.

[9] Florencio, D., and Herley, C. A Large-scale Study of Web Password Habits. In *Proceedings of the 16th International Conference on World Wide Web* (New York, NY, USA, 2007), WWW '07, ACM, pp. 657–666.

[10] Fluhrer, S. R., and McGrew, D. Statistical analysis of the alleged RC4 keystream generator. In *FSE* (2000), B. Schneier, Ed., vol. 1978 of *Lecture Notes in Computer Science*, Springer, pp. 19–30.

[11] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L. HTTP Authentication: Basic and Digest Access authentication. RFC 2617 (Informational), June 1999.

[12] Garman, C., Paterson, K. G., and van der Merwe, T. Attacks only get better: Password recovery attacks against RC4 in TLS. Full version of this paper. Available from `http://www.isg.rhul.ac.uk/tls/RC4mustdie.html`.

[13] Isobe, T., Ohigashi, T., Watanabe, Y., and Morii, M. Full plaintext recovery attack on broadcast RC4. In *Preproceedings of FSE* (2013).

[14] Mantin, I. Predicting and distinguishing attacks on RC4 keystream generator. In *EUROCRYPT* (2005), R. Cramer, Ed., vol. 3494 of *Lecture Notes in Computer Science*, Springer, pp. 491–506.

[15] Mantin, I., and Shamir, A. A practical attack on broadcast RC4. In *FSE* (2001), M. Matsui, Ed., vol. 2355 of *Lecture Notes in Computer Science*, Springer, pp. 152–164.

[16] Ohigashi, T., Isobe, T., Watanabe, Y., and Morii, M. How to recover any byte of plaintext on RC4. In *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers* (2013), T. Lange, K. E. Lauter, and P. Lisonek, Eds., vol. 8282 of *Lecture Notes in Computer Science*, Springer, pp. 155–173.

[17] Salowey, J., Zhou, H., Eronen, P., and Tschofenig, H. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077 (Proposed Standard), Jan. 2008.

[18] Sarkar, S., Sen Gupta, S., Paul, G., and Maitra, S. Proving TLS-attack related open biases of RC4. *IACR Cryptology ePrint Archive 2013* (2013), 502.

[19] Sen Gupta, S., Maitra, S., Paul, G., and Sarkar, S. (Non-) random sequences from (non-) random permutations – analysis of RC4 stream cipher. *Journal of Cryptology 27*, 1 (2012), 67–108.

[20] Weir, M., Aggarwal, S., Collins, M. P., and Stern, H. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010* (2010), E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, Eds., ACM, pp. 162–175.

[21] Yan, J., Blackwell, A., Anderson, R., and Grant, A. Password Memorability and Security: Empirical Results. *IEEE Security and Privacy 2*, 5 (Sept. 2004), 25–31.

[22] Zviran, M., and Haga, W. J. Password Security: An Empirical Study. *J. Manage. Inf. Syst. 15*, 4 (Mar. 1999), 161–185.