Big Bias Hunting in Amazonia: Large-scale Computation and Exploitation of RC4 Biases (Invited Paper)

Kenneth G. Paterson[1], Bertram Poettering[1], and Jacob C. N. Schuldt[2]

[1] Information Security Group, Royal Holloway, University of London
[2] Research Institute for Secure Systems, AIST, Japan

Abstract. RC4 is (still) a very widely-used stream cipher. Previous work by AlFardan *et al.* (USENIX Security 2013) and Paterson *et al.* (FSE 2014) exploited the presence of biases in the RC4 keystreams to mount plaintext recovery attacks against TLS-RC4 and WPA/TKIP. We improve on the latter work by performing large-scale computations to obtain accurate estimates of the single-byte and double-byte distributions in the early portions of RC4 keystreams for the WPA/TKIP context and by then using these distributions in a novel variant of the previous plaintext recovery attacks. The distribution computations were conducted using the Amazon EC2 cloud computing infrastructure and involved the coordination of 2^{13} hyper-threaded cores running in parallel over a period of several days. We report on our experiences of computing at this scale using commercial cloud services. We also study Microsoft's Point-to-Point Encryption protocol and its use of RC4, showing that it is also vulnerable to our attack techniques.

Keywords: RC4, plaintext recovery attack, WPA, TKIP, MPPE

1 Introduction

1.1 RC4 and its Applications

The stream cipher RC4, originally designed by Ron Rivest, is a beautifully compact and fast algorithm. It became public in 1994 and has since been applied in a very wide variety of secure communications protocols, including SSL/TLS (as analysed in [1,7,13,16]); WEP [5] (where its particular usage led to devastating attacks including complete, efficient key recovery, see [20] for a summary and references); WPA [6] (as analysed in [21,20,22,15,18]); Microsoft's Point-to-Point Encryption protocol [14] (MPPE, as analysed here); and some Kerberos-related encryption modes [8]. A selection of additional, non-protocol specific analyses of RC4 can be found in [3,2,12,11,10,19].

Of particular relevance for this work are the results of AlFardan *et al.* [1]. They introduced a simple, Bayesian statistical method that recovers plaintexts that are repeatedly encrypted under RC4 by exploiting biases in RC4 keystreams.

Their approach was successfully applied to RC4 in HTTPS (i.e., HTTP over SSL/TLS), where a fresh pseudorandom 128-bit key is used for each SSL/TLS connection, and where the repeated encryption of HTTP cookies can be arranged by having malicious JavaScript running in the target user's browser.

1.2 RC4 in WPA/TKIP

The work of AlFardan *et al.* motivated us to explore RC4's usage in other deployed protocols, in an attempt to determine whether similar weaknesses exist and are exploitable. Our first focus was the wireless network encryption protocol WPA/TKIP [6], with results presented in [15]. While WPA/TKIP was only ever intended as a stop-gap to replace WEP until stronger cryptography could be deployed, a recent survey [22] showed that it is still in widespread use.

In WPA/TKIP, fresh 16-byte (128-bit) RC4 keys are used for every frame transmitted on the wireless network, but the first three bytes of the key are determined by two bytes $\overline{\mathtt{TSC}} = (\mathtt{TSC}_0, \mathtt{TSC}_1)$ of a public value, $\mathtt{TSC}$, which increments on a frame-by-frame basis; the remaining 13 bytes of the per-frame key are generated pseudorandomly. As observed in [15] and independently in [18], the dependence of the RC4 key on $\overline{\mathtt{TSC}}$ in turn induces large, $\overline{\mathtt{TSC}}$-dependent, single-byte biases in the initial positions of RC4 keystreams. This suggests the attack proposed in [15]: bin the available ciphertexts into 2^{16} bins, one bin for each possible value of $\overline{\mathtt{TSC}}$; perform a Bayesian analysis as per [1] for each bin; and then combine the results across all the bins to estimate the likelihood for each plaintext byte candidate. But this attack requires the computation of accurate single-byte distributions for RC4 keystreams for each of the 2^{16} values of $\overline{\mathtt{TSC}}$. We estimated in [15] that the analysis of 2^{32} – 2^{40} RC4 keystreams per $\overline{\mathtt{TSC}}$ would be needed to achieve sufficient accuracy, for a total of 2^{48} – 2^{56} RC4 keystreams. At that time, this was well beyond our computational capabilities. We resorted to working with 2^{24} keystream per $\overline{\mathtt{TSC}}$ and using the sub-optimal procedure of examining the dependence of the RC4 keystream only on $\mathtt{TSC}_1$, in effect aggregating over $\mathtt{TSC}_0$ (since our intuition was that this byte would have a greater influence in determining the distribution than $\mathtt{TSC}_0$).

Another avenue left unexplored in [15] for WPA/TKIP was the use of double-byte biases in plaintext recovery attacks. Such biases concern the distribution of adjacent pairs of keystream bytes. They were used in [1] in the preferred attack against SSL/TLS, because these biases are persistent throughout the RC4 keystream (whereas the single-byte biases disappear shortly after position 256) and, in the considered attack scenario, it was not possible to arrange for the target plaintext bytes (an HTTP cookie) to appear sufficiently early in the sequence of plaintext bytes. It's also possible that using a double-byte bias attack would improve plaintext recovery rates in the early positions. To extend the double-byte bias attack of [15] to the WPA/TKIP setting would then require the computation of the double-byte keystream distributions, ideally on a per-$\overline{\mathtt{TSC}}$ basis. This would not only require enormous numbers of keystreams to obtain sufficient accuracy, but also significant storage: just to describe the double-byte distribution per position and $\overline{\mathtt{TSC}}$ requires 2^{16} numbers, each typically 32 bits in size, leading to a total storage requirement of 8 Terabytes just to record the double-byte distributions for the first 512 keystream positions.

1.3 RC4 in MPPE

Microsoft's Point-to-Point Encryption (MPPE), as specified in [14,23], is a venerable security protocol that can be used on top of the Point-to-Point Tunnelling Protocol (PPTP). The latter is itself a general-purpose protocol encapsulation method that is commonly used for providing Virtual Private Networking services to devices running Microsoft operating systems, including Windows 8 and the Windows Server family of products.

MPPE uses RC4 with a non-standard method for selecting keys. For example, when a 40-bit key is used, MPPE starts with an 8-byte key $\mathtt{K} = (\mathtt{K}_0, \ldots, \mathtt{K}_7)$ that is itself derived by hashing a user password, an authentication protocol challenge, and other public information. MPPE then sets $\mathtt{K}_0 = \mathtt{0xD1}, \mathtt{K}_1 = \mathtt{0x26}, \mathtt{K}_2 = \mathtt{0x9E}$. It is then natural to ask: does this method for selecting keys in MPPE lead to a different bias structure in its RC4 keystreams, and does this help or hinder plaintext recovery attacks akin to those of [1]?

1.4 Our Contributions and Paper Organisation

Section 2 provides further background on the RC4 stream cipher and its use in WPA and MPPE.

In Section 3, we report on our computations of more refined, per-$\overline{\mathtt{TSC}}$, single-byte and double-byte RC4 keystream distributions for WPA/TKIP. In slightly more detail, we computed these distributions for the first 512 keystream bytes, based on 2^{48} keys for the single-byte case and 2^{46} keys for the double-byte case. We made use of the Amazon Elastic Compute Cloud (Amazon EC2)[3], which is part of Amazon Web Services, to perform the computations. We used approximately 30 virtual-core-years for the single-byte computation and 33 virtual core-years for the double-byte computation. Since, to us, a total of 63 virtual-core-years was quite a significant amount of computation (costing roughly US$41k[4]) and because we faced a number of obstacles in working at this scale, we report in some detail on our experiences of working with Amazon EC2. One notable feature revealed by our large-scale computations is the presence of $\overline{\mathtt{TSC}}$-dependent, single-byte biases well beyond position 256 in the RC4 keystream.

Section 4 describes a plaintext recovery attack on WPA/TKIP that exploits our newly-computed and more accurate single-byte distributions for RC4 keystreams, comparing it to our previous results from [15].

Section 5 describes a novel plaintext recovery attack on WPA/TKIP that exploits per-$\overline{\mathtt{TSC}}$, double-byte biases in RC4 keystreams. This attack combines the double-byte bias attack from [1] with the idea of binning that was developed for the case of single-byte biases in [15].

[3] http://aws.amazon.com/ec2/

[4] Here, and throughout, we quote prices exclusive of sales taxes at 20%.

Algorithm 1: RC4 KSA

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

Algorithm 2: RC4 PRGA

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

Fig. 1. Algorithms implementing the RC4 stream cipher. All additions are performed modulo 256.

In Section 6, we report on the single-byte keystream distributions for RC4 when it is keyed according to the MPPE specification. In short, we found the distributions to be highly skewed and amenable to exploitation using our attack techniques.

Finally, Section 7 presents our conclusions and remarks on open problems.

2 Further Background

2.1 The RC4 Stream Cipher

Technically, RC4 consists of two algorithms: a *key scheduling algorithm* (KSA) and a *pseudo-random generation algorithm* (PRGA), which are specified in Algorithms 1 and 2. The KSA takes as input a key $\mathtt{K}$, typically a byte-array of length between 5 and 32 (i.e., 40 to 256 bits), and produces the initial internal state $st_0 = (i, j, \mathcal{S})$, where $\mathcal{S}$ is the canonical representation of a permutation on the set $[0, 255]$ as an array of bytes, and i, j are indices into this array. The PRGA will, given an internal state st_r, output 'the next' keystream byte Z_{r+1}, together with the updated internal state st_{r+1}.

2.2 WPA/TKIP

A detailed description of how RC4 is used in the WPA/TKIP context is given in [15]. In short, WPA/TKIP generates a fresh 128-bit key $\mathtt{K} = (\mathtt{K}_0, \ldots, \mathtt{K}_{15})$ for RC4 for each frame that is transmitted; the key is a function of the temporal encryption key $\mathtt{TK}$ (128 bits), the TKIP sequence counter $\mathtt{TSC}$ (48 bits), and the transmitter address $\mathtt{TA}$ (48 bits). A single value of $\mathtt{TK}$ is used over many frames, while $\mathtt{TSC}$ increments from frame to frame; meanwhile $\mathtt{TA}$ is fixed. Very importantly, the function used to compute $\mathtt{K}$ adds a specific structure added to "preclude the use of known RC4 weak keys" [6]. More precisely, writing $\mathtt{TSC} = (\mathtt{TSC}_0, \mathtt{TSC}_1, \ldots, \mathtt{TSC}_5)$, we have

$$\mathtt{K}_0 = \mathtt{TSC}_1 \qquad \mathtt{K}_1 = (\mathtt{TSC}_1 \mid \mathtt{0x20}) \ \& \ \mathtt{0x7f} \qquad \mathtt{K}_2 = \mathtt{TSC}_0 \tag{1}$$

while $\mathtt{K}_3, \ldots, \mathtt{K}_{15}$ can be considered to be pseudorandom functions of $\mathtt{TK}$, $\mathtt{TSC}$ and $\mathtt{TA}$. Notably here, bytes $\mathtt{K}_0, \mathtt{K}_1, \mathtt{K}_2$ depend only on bytes $\mathtt{TSC}_0$ and $\mathtt{TSC}_1$ of $\mathtt{TSC}$. Moreover, the bits of $\mathtt{TSC}_1$ are used twice. So the bytes of $\mathtt{K}$ have more structure than they would if they were chosen with uniform distribution. The per-frame key $\mathtt{K}$ is then used to produce an RC4 keystream, following our description of RC4 above. The TKIP plaintext (consisting of the frame payload, a 64-bit MAC value `MIC` , and a 32-bit Integrity Check Vector `ICV`) is then XORed in a byte-by-byte fashion with the RC4 keystream.

2.3 MPPE

MPPE provides a confidentiality service over PPTP using the RC4 algorithm. Keys for the RC4 algorithm come from a separate authentication and key establishment protocol, such as MS-CHAPv1, MS-CHAPv2 or EAP-TLS; the first two of these were broken in [17] and [4], respectively, leading to the deprecation of the first and the recommendation only to use the second with additional protection from PEAP[5].

RFC 3079 [23] describes in detail how the keys used in MPPE's instantiation of RC4 are derived from preceding authentication and key establishment protocols. Three different RC4 key lengths are supported, according to [23]: 40-bit, 56-bit and 128-bit. When a 40-bit key is used, MPPE starts with an 8-byte key $\mathtt{K} = (\mathtt{K}_0, \ldots, \mathtt{K}_7)$ that is itself derived by hashing the password, the authentication protocol challenge, and other public information. MPPE then overwrites $\mathtt{K}_0 = \mathtt{0xD1}, \mathtt{K}_1 = \mathtt{0x26}, \mathtt{K}_2 = \mathtt{0x9E}$. When a 56-bit key is used, the protocol starts with the same 8-byte key and then sets $\mathtt{K}_0 = \mathtt{0xD1}$; when a 128-bit key is used, a similar procedure involving password and challenge hashing is used to generate a 16-byte key $\mathtt{K}$, and no bytes of $\mathtt{K}$ are overwritten.

Furthermore, MPPE operates in two modes, with the mode in use being determined by a PPTP header field. In stateless mode, the RC4 key is refreshed and the cipher restarted for each PPTP packet sent. By contrast, in stateful mode, the RC4 key is refreshed only every 256 packets. In both cases, refreshing the key involves hashing the old key with the first key for the session (called `StartKey` in [14]) to generate a value `InterimKey`, then an RC4 encryption step in which `InterimKey` is used to encrypt itself to generate a key $\mathtt{K}$ of either 8 or 16 bytes, and finally setting bytes as described above. See [14, Section 7] for details.

From the above description it may be remarked that, while the hashing and encryption steps used in deriving the RC4 keys may be intended to render them

[5] https://technet.microsoft.com/library/security/2743314

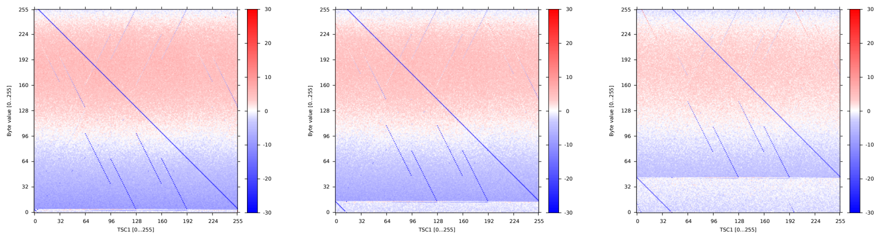
pseudorandom, in the 40-bit and 56-bit cases, they have additional structure that may be expected to lead to additional and/or different biases in the RC4 keystream as compared to the 128-bit case.Further, the use of stateless mode would mean a fresh RC4 key (with additional structure in the 40-bit and 56-bit cases) for every packet sent. These observations mean that MPPE in stateless mode can be expected to be vulnerable to plaintext recovery attacks similar to those developed in [1,15]. Since the protocol encapsulated by MPPE is likely to be IP, similar fields as those identified in [15] could be targeted. We note that while key lengths of 40 and 56 bits are small enough that a simple brute-force search might initially seem to be more efficient than mounting a bias analysis using our techniques, in the stateless case, such a brute-force search would only recover the key used for a single packet. Moreover, a basic analysis suggests that a 2^{64} attack would be needed to recover `StartKey` from which all keys in the session are derived. So our approach may be an attractive alternative if specific plaintext bytes are targeted for recovery.

3 Large-Scale Computation of RC4 Keystream Distributions for WPA/TKIP Keys

3.1 Computing Keystream Distributions and Finding New Biases

As noted in the introduction, in our previous work on WPA/TKIP in [15], we worked with a total of only 2^{40} keystreams and only with single-byte distributions for the first 256 positions. In an effort to further improve our attacks, we decided to perform larger-scale computations using, in addition to our own local resources, the Amazon EC2 cloud computing infrastructure to estimate both the single-byte and double-byte keystream distributions for the first 512 positions, on a per-$\overline{\texttt{TSC}}$ basis.

Because the double-byte biases are smaller than the single-byte ones (typically by a factor of roughly 2^8), many more keystreams would be needed to accurately estimate double-byte distributions than for single-byte ones. However, we chose to focus our effort on the single-byte case here, computing distributions based on 2^{32} keystreams per $\overline{\texttt{TSC}}$ in the single-byte case and based on 2^{30} keystreams per $\overline{\texttt{TSC}}$ in the double-byte case. The reasons for this focus are as follows. Using our local computational resources, we determined that it would be difficult to use the full per-$\overline{\texttt{TSC}}$ distributions in a double-byte attack akin to that of [1] because of the complexity of handling so much data when running attacks (for example, we would need to deal with 16GB of distribution data and perform 2^{48} multiplications of real numbers to analyse a single byte position). Rather, an aggregated approach seemed more likely to be feasible for the double-byte setting. Here our idea was to start with 2^{30} keystreams per $\overline{\texttt{TSC}}$ and combine the 2^8 distributions for each $\texttt{TSC}_1$ value (called $\texttt{TSC}_0$-aggregation in [15]) to obtain 2^8 different double-byte distributions, one per $\texttt{TSC}_1$-value, each distribution now based on 2^{38} keystreams. This not only boosts the number of keystreams per distribution estimate (good for accurately estimating biases), but also reduces the



(a) Position 260 (b) Position 270 (c) Position 300

Fig. 2. Pictorial representation of biases in RC4 keystreams for random $\mathtt{TSC}_0$-aggregated WPA/TKIP keys at keystream positions 260, 270, and 300, for different $\mathtt{TSC}_1$ values (x-axis) and byte values (y-axis). At each point we encode the bias in the keystream for the ($\mathtt{TSC}_1$,value) combination as a colour; precisely, we encode the difference between the occurring probability and the (expected) probability 1/256, scaled up by a factor of 2^{24}, capped to values in $[-30, +30]$.

size of the distribution data and computation both by a factor of 256 (making simulation of attacks much more feasible).

Our computations went well beyond those of prior RC4 cryptanalyses in scale (e.g., [1,15]), and indeed we were rewarded by discovering new $\mathtt{TSC}_1$-dependent single-byte biases in positions all the way up to 512 (see Figure 2 for examples at specific positions). The existence of these biases is surprising in view of the behaviour of single-byte biases observed in previous works and, in principle, would allow the recovery of plaintext using a single-byte attack like that presented in [15] and Section 4 below. It is an open problem to determine how far into the RC4 keystream these biases persist.

3.2 Reflections on Using Amazon EC2

The task of computing accurate estimates of RC4 keystream distributions is well-suited to distributed computation. In particular, in the case of WPA, the probability distribution for each $\overline{\mathtt{TSC}}$ value can be estimated independently by generating keystreams using randomly chosen WPA keys for that $\overline{\mathtt{TSC}}$ (having the structure described in Section 2.2). This makes performing the computation using cloud services such as Amazon EC2 seem appealing, on account of its virtually unlimited computing capacity being able to provide the computational resources required to complete the computation within an acceptable period of time.

For our computation of the per-$\overline{\mathtt{TSC}}$ bias estimates, we used Amazon EC2 to create 256 virtual machines of the type ‘c3.x8large’, each providing 32 ‘virtual’ cores. The underlying hardware of the virtual machines were servers equipped with Intel Xeon 2.8GHz processors. Note, however, that each of the cores of a virtual machine corresponds to a hyper-threaded core of the underlying CPU i.e., one ‘c3.x8large’ instance effectively corresponds to a machine with 16 physical cores. To manage the virtual machines, we utilized boto[6] which implements a Python interface to Amazon EC2. This provided a simple and straightforward way to automate management and access to the virtual machines, and made it relatively easy to set up the execution of the computation using a combination of Python and shell scripts. The virtual machines were all initialized with an Ubuntu 13.10 image obtained through the AWS Marketplace[7].

Each virtual machine was set up to compute the keystream distributions for all $\texttt{TSC}_0$ values given a fixed $\texttt{TSC}_1$ value, and to split this computation equally among the 32 available virtual cores. To make the WPA keystream generation efficient, we used the RC4 implementation in OpenSSL[8]. However, experiments showed that to reach the desired number of keystreams with our available budget, further optimizations were required. Additional experiments revealed that the amount of available cache in the underlying CPUs on which the virtual machines were running, and how this cache was utilized, played an important role in the performance of the keystream distribution generation. Specifically, the chance of cache misses occurring when updating the keystream distribution statistics was found to have a large influence on performance.

To address this, we used a combination of two different approaches to reduce the chance of cache misses occurring. Firstly, to fit the array storing the counters used to collect the statistics of the keystream distribution into the cache memory, we "packed" multiple small-width counters into single 64-bit integers and implemented logic for handling counter overflows. Secondly, instead of updating the keystream distribution statistics after each keystream has been generated, we stored multiple keystreams in memory before updating the statistics. This implies that multiple updates of the statistics for a single position can be done sequentially, which, assuming the appropriate memory layout, increases the chance of a cache hit. While these optimizations only provided small gains for the computation of single-byte biases, significant gains were achieved for the computation of double-byte biases.

Single-byte computations Using the above setup, each virtual core was capable of generating and processing on average 294k length 512 WPA keystreams per second for single-byte distributions. Hence, computing the per-$\overline{\texttt{TSC}}$ single-byte distributions based on 2^{32} keystreams for each $\overline{\texttt{TSC}}$ value (i.e., 2^{48} keystreams in total), took 9.56×10^8 virtual core seconds in total, or approximately 30 virtual core years. Due to the large degree of parallelism in our setup, this corresponds to an actual running time of slightly more than 32 hours.

While each of the 256 virtual machine was set up identically, a single virtual machine ran significantly slower than the others, and was only capable of processing approximately 180k keystreams per second. We suspect that other virtual machines running on the same underlying hardware might have affected

[6] http://github.com/boto/boto
[7] http://aws.amazon.com/marketplace/
[8] https://www.openssl.org/

the performance of this virtual machine. Due to this issue, it took approximately 52 hours to complete the computation of the single-byte distributions.

At the time we did the experiments, the cost of running a single "c3.x8large" virtual machine instance was US$2.40 per hour, leading to a cost of US$614 per hour when running all 256 instances simultaneously.

To store the generated keystream distributions, we attached a separate Amazon Elastic Block Storage (EBS) volume to each virtual machine. This gave us the option of terminating a virtual machine without erasing the generated data, and furthermore allowed us to use a single virtual machine to inspect and process all generated data, by sequentially attaching the EBS volumes to this machine. The latter provided a more cost effective solution than running the virtual machines in parallel, and a faster solution than resuming each virtual machine sequentially. We stored the distribution for each $\overline{\texttt{TSC}}$ value as a sequence of binary encoded 32-bit integers, leading to a storage requirement of 512KB per distribution (128MB per virtual machine), or 32GB in total. However, since the minimum size of an EBS volume is 1GB, we allocated a total of 256GB of EBS storage (note that a single EBS volume cannot be mounted by multiple virtual machines simultaneously). The cost of EBS storage was US$0.05 per GB per month, leading to a cost of just US$12.60 a month to maintain the EBS volumes.

Double-byte computations Working with double-byte keystream distributions introduced significant overheads compared to the single-byte case, both in terms of computation and storage. With the previously mentioned optimizations, each virtual core was capable of processing on average 67k WPA keystreams per second. Hence, computing the per-$\overline{\texttt{TSC}}$ double-byte keystream distributions based on 2^{30} keystreams for each $\overline{\texttt{TSC}}$ value (i.e., 2^{46} keystreams in total), took 1.05×10^9 virtual core seconds in total, or approximately 33 virtual core years. In our setup, this corresponds to an actual running time of slightly more than 34 hours, but due to the virtual machines being sequentially initialized and an issue with a single virtual machine, the time it took to complete the computation was approximately 48 hours. More specifically, the issue that arose was that the virtual machine in question was reset and rebooted during the computations, and hence did not complete its assigned task. We were unable to identify the cause of this event, and simply restarted the relevant computations manually.

As for the single-byte distributions, we created separate EBS volumes to store the double-byte distributions. However, each $\overline{\texttt{TSC}}$-specific double-byte distribution requires 128MB of storage when stored as a sequence of 32-bit integers, leading to a storage requirement of 32GB per virtual machine, or 8TB in total. This increased storage overhead not only led to an increased cost (US$410 per month), but also created additional practical issues which we had to handle. For example, since the EBS volumes are implemented via network attached storage (NAS), writing the distribution data to an EBS volume caused a significant delay in some instances. In particular, we observed that immediately after completion of the keystream distribution generation, detaching an EBS volume might not succeed, which in turn could interrupt the shutdown of a virtual machine. Furthermore, making all data available to a single machine at the same time, which is required to efficiently run attack simulations, was made more difficult by the 8TB size of the dataset. We decided to transfer the complete dataset to our local storage array both to run the attack simulations and to permanently store the data. For this purpose, we used bbcp[9], which is capable of transferring large amounts of data between network computers using multiple TCP streams and large transfer windows, and allowed us to obtain a transfer speed of approximately 50MB per second, leading to a total transfer time of slightly more than 48 hours. Note that data transfers out of Amazon EC2 were charged at US$0.12 per GB, resulting in a US$983 cost to move the complete 8TB dataset to our local storage.

Our experience of using Amazon EC2 to compute estimates of the per-$\overline{\texttt{TSC}}$ biases suggests that Amazon provides a flexible platform which is well suited to perform this type of computation, and that the practical difficulties arising in the distribution of the computation can be overcome with moderate effort.

4 Plaintext Recovery Attacks Against WPA/TKIP Based on Single-Byte Biases

4.1 The Attack of Paterson, Poettering and Schuldt[15]

The attack against WPA/TKIP in [15] builds on the single-byte bias attack (on TLS) of [1]. Both attacks work for the setting where the same plaintext is encrypted many times under different RC4 keys to obtain a set of ciphertexts. The key idea of both attacks is that, in any given position r of the ciphertext stream, a guess for the repeated plaintext byte in that position induces a distribution on the keystream in position r, via XORing the guess with byte r in each of the ciphertexts in turn. This induced distribution can be compared to the known distribution in keystream position r (which is obtained by sampling), and the choice of plaintext guess giving the "best fit" selected as the attack's output for position r. This is formalised as a Bayesian procedure, leading to the output in position r as being the plaintext candidate that maximises the probability of observing the induced keystream distribution in position r.

The innovation in [15] (and independently observed in [18]) was to recognise that in WPA/TKIP a different keystream distribution can – and should – be used for each value of the byte pair $\overline{\texttt{TSC}} = (\texttt{TSC}_0, \texttt{TSC}_1)$ when estimating the probabilities of the induced keystream distributions. This leads to an algorithm that "bins" ciphertexts into 2^{16} groups, one group per $\overline{\texttt{TSC}}$, computes the induced keystream probability for each group, and takes the product of these across the groups to compute the probabilities for each plaintext candidate. Since our new double-byte algorithm in Section 5 can be seen as an extension of our algorithm in [15], we explain the latter here in more detail.

[9] http://www.slac.stanford.edu/~abh/bbcp/

We first obtain a detailed picture of the distributions of RC4 keystream bytes Z_r, for all positions r in some range, on a per $(\mathtt{TSC}_0, \mathtt{TSC}_1)$ pair basis, by gathering statistics from keystreams generated using a large number of random keys. That is, for all r in our selected range, we estimate

$$p_{\overline{\mathtt{TSC}},r,k} := \Pr(Z_r = k)\,, \quad \overline{\mathtt{TSC}} \in \mathcal{T}sc\mathcal{S}p, \quad k \in \mathcal{B}yte,$$

where here (and henceforth) $\mathcal{B}yte$ denotes the set $\{\mathtt{0x00}, \ldots, \mathtt{0xFF}\}$, $\mathcal{T}sc\mathcal{S}p$ denotes the set $\mathcal{B}yte \times \mathcal{B}yte$, and where the probability is taken over the random choice of the RC4 encryption key $\mathtt{K}$, subject to the structure on $\mathtt{K}_0, \mathtt{K}_1, \mathtt{K}_2$ induced by $\overline{\mathtt{TSC}}$.

Now suppose we have S ciphertexts $C_1, \ldots, C_S$ available for our attack. We partition these into 2^{16} groups according to the value of $\overline{\mathtt{TSC}}$ (recall that the $\overline{\mathtt{TSC}}$ value is public); for convenience, we assume the resulting bins of ciphertexts are all of equal size $T = S/2^{16}$. Let the bin of ciphertexts associated with a particular value of $\overline{\mathtt{TSC}}$ be denoted $\mathcal{S}_{\overline{\mathtt{TSC}}}$ and have members $C_{\overline{\mathtt{TSC}},j}$ for $j = 1, \ldots, T$; we denote the byte at position r of $C_{\overline{\mathtt{TSC}},j}$ by $C_{\overline{\mathtt{TSC}},j,r}$. For any position r and any candidate plaintext byte μ for that position, vector $\big(N^{(\mu)}_{\overline{\mathtt{TSC}},r,k}\big)_{k\in\mathcal{B}yte}$ with

$$N^{(\mu)}_{\overline{\mathtt{TSC}},r,k} = |\{j \in [1\,..\,T] \mid C_{\overline{\mathtt{TSC}},j,r} = k \oplus \mu\}| \qquad (\mathtt{0x00} \le k \le \mathtt{0xFF})$$

represents the distribution on Z_r required to obtain the observed ciphertext bytes $(C_{\overline{\mathtt{TSC}},j,r})_{1\le j\le T}$ for bin $\mathcal{S}_{\overline{\mathtt{TSC}}}$ by encrypting μ. The probability $\lambda_{\overline{\mathtt{TSC}},r,\mu}$ that plaintext byte μ is encrypted to bytes $(C_{\overline{\mathtt{TSC}},j,r})_{1\le j\le T}$ in bin $\mathcal{S}_{\overline{\mathtt{TSC}}}$ for position r now follows the distribution:

$$\lambda_{\overline{\mathtt{TSC}},r,\mu} = \prod_{k\in\mathcal{B}yte} \left(p_{\overline{\mathtt{TSC}},r,k}\right)^{N^{(\mu)}_{\overline{\mathtt{TSC}},r,k}} \,. \tag{2}$$

Note that this expression differs from that in [15] by the omission of factorial terms arising in the multinomial distribution. Those terms do not need to be included in the formal Bayesian procedure underlying the attack (since we are interested in the probability of a group of ciphertexts bytes as given in a particular sequence rather than in unordered form). Moreover, their removal makes the attack slightly easier to implement.

Now the probability that plaintext byte μ is encrypted to the vector of bytes $(C_{\overline{\mathtt{TSC}},j,r})_{1\le j\le T}$ across all bins $\mathcal{S}_{\overline{\mathtt{TSC}}}$ in position r can be precisely calculated as

$$\lambda_{r,\mu} = \prod_{\overline{\mathtt{TSC}}\in\mathcal{T}sc\mathcal{S}p} \lambda_{\overline{\mathtt{TSC}},r,\mu} \,.$$

By computing $\lambda_{r,\mu}$ for all $\mu \in \mathcal{B}yte$, and identifying $P^*_r = \mu$ such that $\lambda_{r,\mu}$ is largest, we determine the maximum-likelihood plaintext byte value P^*_r.

Note that, for each position r and group of bytes $(C_{\overline{\mathtt{TSC}},j,r})_{1\le j\le T}$, values $N^{(\mu)}_{\overline{\mathtt{TSC}},r,k}$ can be computed from values $N^{(\mu')}_{\overline{\mathtt{TSC}},r,k}$ by using the equation $N^{(\mu)}_{\overline{\mathtt{TSC}},r,k} = N^{(\mu')}_{\overline{\mathtt{TSC}},r,k\oplus\mu'\oplus\mu}$, for all k. Further, computing and comparing $\log(\lambda_{\overline{\mathtt{TSC}},r,\mu})$ and

Algorithm 3: Plaintext recovery attack using $\overline{\texttt{TSC}}$ binning

```
input : $\{C_{\overline{\texttt{TSC}},j}\}_{\overline{\texttt{TSC}}\in\mathcal{T}sc\mathcal{S}p,1\le j\le T}$ – $S = 2^{16}\cdot T$ independent encryptions of fixed
        plaintext $P$
        $r$ – target byte position
        $(p_{\overline{\texttt{TSC}},r,k})_{\overline{\texttt{TSC}}\in\mathcal{T}sc\mathcal{S}p,k\in\mathcal{B}yte}$ – keystream distributions for all $\overline{\texttt{TSC}}$ at pos. $r$
output: $P_r^*$ – estimate for plaintext byte $P_r$
begin
    $N_{\overline{\texttt{TSC}},k} \leftarrow 0$    for all $\overline{\texttt{TSC}} \in \mathcal{T}sc\mathcal{S}p$, $k \in \mathcal{B}yte$
    for $\overline{\texttt{TSC}} = (\texttt{0x00}, \texttt{0x00})$ to $(\texttt{0xFF}, \texttt{0xFF})$ do
        for $j = 1$ to $T$ do
            $k \leftarrow C_{\overline{\texttt{TSC}},j,r}$
            $N_{\overline{\texttt{TSC}},r,k} \leftarrow N_{\overline{\texttt{TSC}},r,k} + 1$
    for $\overline{\texttt{TSC}} = (\texttt{0x00}, \texttt{0x00})$ to $(\texttt{0xFF}, \texttt{0xFF})$ do
        for $\mu = \texttt{0x00}$ to $\texttt{0xFF}$ do
            for $k = \texttt{0x00}$ to $\texttt{0xFF}$ do
                $N^{(\mu)}_{\overline{\texttt{TSC}},r,k} \leftarrow N_{\overline{\texttt{TSC}},r,k\oplus\mu}$
            $\lambda_{\overline{\texttt{TSC}},r,\mu} \leftarrow \sum_{k\in\mathcal{B}yte} N^{(\mu)}_{\overline{\texttt{TSC}},r,k} \log p_{\overline{\texttt{TSC}},r,k}$
    for $\mu = \texttt{0x00}$ to $\texttt{0xFF}$ do
        $\lambda_{r,\mu} \leftarrow \sum_{\overline{\texttt{TSC}}\in\mathcal{T}sc\mathcal{S}p} \lambda_{\overline{\texttt{TSC}},r,\mu}$
    $P_r^* \leftarrow \arg\max_{\mu\in\mathcal{B}yte} \lambda_{r,\mu}$
    return $P_r^*$
```

$\log(\lambda_{r,\mu})$ instead of $\lambda_{\overline{\texttt{TSC}},r,\mu}$ and $\lambda_{r,\mu}$ makes the computation more efficient and accuracy easier to maintain. Adding these optimisations leads to the attack in Algorithm 3 (which differs from the corresponding attack in [15] only in the omission of a term $F_{\overline{\texttt{TSC}}}$ corresponding to the factorial terms discussed above and some small notational changes).

4.2 Attacks Based on Aggregation

One method of coping with noisy estimates for the probabilities $p_{\overline{\texttt{TSC}},r,k}$ that was extensively explored in [15] was to consider aggregation of distributions over $\texttt{TSC}_0$ or over both $\texttt{TSC}_0$ and $\texttt{TSC}_1$ (effectively increasing the number of keys by factors of 2^8 and 2^{16}, respectively). It is not difficult to see how to modify Algorithm 3 to work with 2^8 bins, one for each value of $\texttt{TSC}_1$, instead of 2^{16} bins. The execution of the modified algorithm becomes in practice faster, since each estimate for a plaintext byte μ now only involves calculation of $\lambda_{\overline{\texttt{TSC}},r,\mu}$ over 2^8 $\texttt{TSC}_1$ values instead of 2^{16} $(\texttt{TSC}_0, \texttt{TSC}_1)$ pair values. Similarly, one can modify the algorithm to work with just a single bin, one for all values of $\texttt{TSC}_0$ and $\texttt{TSC}_1$, in which case we recover the original algorithm of [1], albeit without the unnecessary factorial terms arising from the use of multinomial distributions and using WPA/TKIP-specific distributions in place of the original RC4 distributions reported in [1].

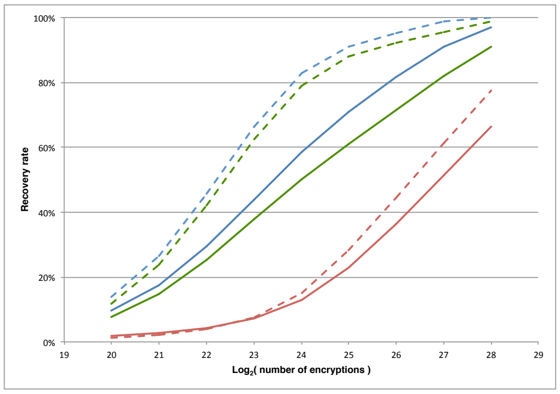


Fig. 3. Average success rates of non-aggregated (blue), $\mathtt{TSC}_0$-aggregated (green), and fully aggregated (red) single-byte plaintext recovery attacks for byte positions 1 to 256 (based on 256 experiments). Punctured lines represent the average recovery rates for the odd byte positions.

However, the cost of using aggregation is that it "throws away" statistical information that may be of use in improving the accuracy of the attack for a given number of ciphertexts S. Indeed, this is demonstrably the case: as we report below, using our new estimates for the probabilities $p_{\overline{\mathtt{TSC}},r,k}$ computed using a total of 2^{48} keystreams in a full binning (non-aggregated) attack leads to an improvement in accuracy.

4.3 Attack Simulation Results

We implemented the single-byte plaintext recovery attack of Algorithm 3 based on the keystream distribution estimates obtained from the Amazon EC2 computations described in Section 3. We furthermore implemented the $\mathtt{TSC}_0$-aggregated and fully aggregated variants of the attack described in Section 4.2. To obtain bias estimates for the latter two attacks, we aggregated the Amazon EC2 data correspondingly, thereby obtaining estimates based on 2^{40} keystream per $\mathtt{TSC}_1$ value, and 2^{48} keystreams, respectively.

The measured success rates of the attacks are shown in Figure 3. We observe that there is a significant difference in the recovery rates between the fully aggregated attack and the two other attacks, the non-aggregated attack being capable of achieving a similar success rate to the fully aggregated attack using almost 16 times fewer ciphertexts. Likewise, the non-aggregated attack clearly improves upon the $\mathtt{TSC}_0$-aggregated attack, albeit not as significantly; the non-aggregated attack requires on average half as many ciphertexts to achieve a similar success rate to the $\mathtt{TSC}_0$-aggregated attack.

In order to investigate the effect of our new and (presumably) more accurate single-byte keystream distributions, we also compared the performance of Algorithm 3 using keystream distribution estimates based on 2^{24} keystreams per $\overline{\mathtt{TSC}}$

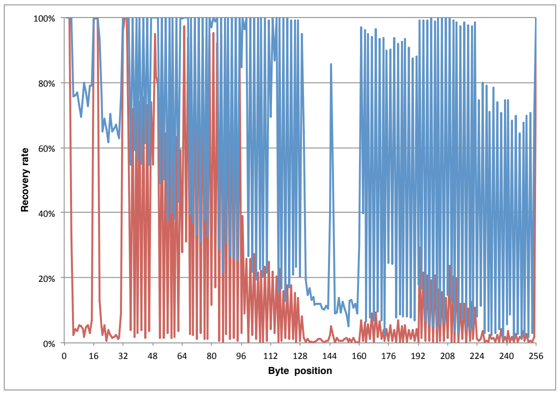


Fig. 4. Success rates of single-byte plaintext recovery attack against TKIP/WPA for positions 1 to 256 with 2^{24} ciphertexts, using keystream distribution estimates based on 2^{24} keystreams (red) and 2^{32} keystreams (blue) per $\overline{\texttt{TSC}}$ (success rates based on 256 experiments).

(as in our previous work [15]) and based on 2^{32} keystreams per $\overline{\texttt{TSC}}$ (obtained from the Amazon EC2 computations described in Section 3). Figure 4 shows the results, with the attacks using the two keystream distributions in combination with 2^{24} ciphertexts in each experiment. There is a clear boost to the success rate of the attack when moving to the refined keystream distribution estimates. The effect is particularly pronounced in the odd positions.

As noted in Section 3, we discovered significant $\texttt{TSC}_1$-dependent, single-byte biases in the RC4 keystreams for WPA/TKIP keys well beyond position 256. The biases are roughly comparable in size to the single-byte biases seen in RC4 keystreams at positions around 250 for random 128-bit keys (as used in TLS and reported in [1]). So we might expect to obtain reliable plaintext recovery with around $2^{30} - 2^{32}$ ciphertexts as in [1]. The full investigation of this avenue is left to future work.

5 Plaintext Recovery Attacks for WPA/TKIP Based on Double-Byte Biases

Our double-byte bias attack against WPA/TKIP builds on the attack in [1], and works in the same setting as the above described single-byte bias attack: the same plaintext is assumed to be encrypted many times under different RC4 keys, yielding a set of ciphertexts which is given as input to the attack algorithm. However, as opposed to the attack based on single-byte biases, the attack does not estimate the likelihoods of the individual plaintext bytes (or plaintext byte pairs). Instead, the basic idea of the attack is to estimate likelihoods of *sequences* of plaintext bytes by considering chains of overlapping plaintext byte pairs in combination with the double-byte biases in the keystream.

More precisely, the attack will construct likelihood estimates for sequences of plaintext bytes that are gradually increasing in length by extending already established sequences and their corresponding likelihood estimates. This is done as follows: consider a sequence of plaintext bytes with an already estimated likelihood, and a candidate for the next plaintext byte in the sequence. By XORing the pair consisting of the last plaintext byte of the existing sequence and the new candidate plaintext byte with the ciphertext byte pairs for the corresponding positions, an induced distribution on the keystream byte pairs is obtained. By comparing this to the known double-byte keystream distribution, a likelihood estimate for the new candidate plaintext byte can be computed; combining this with the likelihood estimate for the initial plaintext sequence, a likelihood estimate for the extended sequence can be obtained.

Note that, using a naive algorithm, the complexity of computing the likelihood estimates for all possible plaintext sequences would grow exponentially in the length of the sequences. Furthermore, considering all possible candidates for the next plaintext byte, but only maintaining a small set of the most likely sequences after each extension, is not guaranteed to produce a plaintext byte sequence that maximises the value of the estimated likelihood. However, as highlighted in [1], by tracking which sequences produce the maximum value for the estimated likelihood for each possible value of the last byte in the sequence, the overall plaintext sequence which maximises the likelihood estimate is guaranteed to be found.

Compared to the algorithm from [1], the algorithm presented here provides two refinements made possible by the specific way RC4 is used in WPA/TKIP. Firstly, as in the attack described in Section 4, likelihood estimates are computed on a per-$\overline{\texttt{TSC}}$ basis, and combined across all $\overline{\texttt{TSC}}$ values to obtain improved overall likelihood estimates. Secondly, the attack not only exploits the per-$\overline{\texttt{TSC}}$ double-byte biases in the WPA keystream, but also takes into account the single-byte biases in the computation of the likelihood estimates. A more detailed description of the algorithm is given next.

To run the algorithm, accurate estimates of both the single-byte and double-byte keystream distributions are required for all positions r the attack is targeting. By considering the statistics gathered by generating a large number of keystreams, we estimate

$$p_{\overline{\texttt{TSC}},r,k} := \Pr(Z_r = k), \quad \text{and} \quad \tilde{p}_{\overline{\texttt{TSC}},r,k_1,k_2} := \Pr(Z_r = k_1 \wedge Z_{r+1} = k_2)$$

where $\overline{\texttt{TSC}} \in \mathcal{T}sc\mathcal{S}p$, $k, k_1, k_2 \in \mathcal{B}yte$, and the probability is taken over a random choice of RC4 key subject to the structure on $\texttt{K}_0$, $\texttt{K}_1$, $\texttt{K}_2$ induced by $\overline{\texttt{TSC}}$.

As in the single-byte bias attack, we suppose we have S ciphertexts available for our attack, and that, when grouped according to $\overline{\texttt{TSC}}$ values, each group contains exactly $T = S/2^{16}$ ciphertexts. We likewise use the notation $C_{\overline{\texttt{TSC}},j,r}$ to denote the ciphertext byte at position r in the jth member of the group of ciphertexts for the value $\overline{\texttt{TSC}}$.

For a given position r, we can now use a similar approach to the single-byte bias attack to compute the likelihood of a candidate byte μ or a candidate byte pair (μ, μ') (at position $(r, r+1)$) corresponding to the encrypted plaintext byte or byte pair. More specifically, the vectors $\big(N^{(\mu)}_{\overline{\texttt{TSC}},r,k_1}\big)_{k_1 \in \mathcal{B}yte}$ and $\big(\tilde{N}^{(\mu,\mu')}_{\overline{\texttt{TSC}},r,k_1,k_2}\big)_{k_1,k_2 \in \mathcal{B}yte}$, where

$$\begin{aligned} N^{(\mu)}_{\overline{\texttt{TSC}},r,k_1} &= |\{j \in [1 \mathinner{..} T] \mid C_{\overline{\texttt{TSC}},j,r} = k_1 \oplus \mu\}| \\ \tilde{N}^{(\mu,\mu')}_{\overline{\texttt{TSC}},r,k_1,k_2} &= |\{j \in [1 \mathinner{..} T] \mid (C_{\overline{\texttt{TSC}},j,r}, C_{\overline{\texttt{TSC}},j,r+1}) = (k_1 \oplus \mu, k_2 \oplus \mu')\}| \ , \end{aligned}$$

represent induced distributions on the keystream byte Z_r and keystream byte pair (Z_r, Z_{r+1}), respectively. Indeed, the probability that plaintext byte μ is encrypted at position r, which we will denote $\alpha_r^{(\mu)}$, and the probability that (μ, μ') is encrypted at position $(r, r+1)$, which we will denote $\beta_r^{(\mu,\mu')}$, can be computed as:

$$\begin{aligned} \alpha_r^{(\mu)} &= \prod_{\overline{\texttt{TSC}} \in \mathcal{T}sc\mathcal{S}p} \prod_{k_1 \in \mathcal{B}yte} \left(p_{\overline{\texttt{TSC}},r,k_1}\right)^{N^{(\mu)}_{\overline{\texttt{TSC}},r,k_1}} , \\ \beta_r^{(\mu,\mu')} &= \prod_{\overline{\texttt{TSC}} \in \mathcal{T}sc\mathcal{S}p} \prod_{k_1,k_2 \in \mathcal{B}yte} \left(\tilde{p}_{\overline{\texttt{TSC}},r,k_1,k_2}\right)^{\tilde{N}^{(\mu,\mu')}_{\overline{\texttt{TSC}},r,k_1,k_2}} . \end{aligned}$$

However, as highlighted earlier, instead of using the above probabilities for individual plaintext byte and byte pairs directly, we use these to construct likelihood estimates for longer sequences of plaintext bytes by considering chains of overlapping byte pairs. More specifically, consider a plaintext byte sequence $\mu_1 \parallel \cdots \parallel \mu_r$ for positions 1 to r with an already established likelihood estimate $\lambda_{\mu_1 \parallel \cdots \parallel \mu_r}$, and a plaintext candidate byte μ_{r+1} for position $r+1$. Then we estimate the likelihood of the plaintext byte sequence $\mu_1 \parallel \cdots \parallel \mu_{r+1}$ as:

$$\lambda_{\mu_1 \parallel \cdots \parallel \mu_{r+1}} = \delta_r^{(\mu_r,\mu_{r+1})} \cdot \lambda_{\mu_1 \parallel \cdots \parallel \mu_r} \tag{3}$$

where $\delta_r^{(\mu_r,\mu_{r+1})}$ denotes the conditional probability that μ_{r+1} is the plaintext byte at position $r+1$ given that the plaintext byte at position r is μ_r. Note that, by the definition of conditional probability, we can compute $\delta_r^{(\mu_r,\mu_{r+1})}$ based on the estimates $\alpha_r^{(\mu_r)}$ and $\beta_r^{(\mu_r,\mu_{r+1})}$ as

$$\delta_r^{(\mu_r,\mu_{r+1})} = \beta_r^{(\mu_r,\mu_{r+1})} / \alpha_r^{(\mu_r)}.$$

In the description of the attack algorithm presented here, it is assumed that the plaintext byte P_1^* at position $r = 1$ is known. This serves as a starting point for the algorithm, i.e., the algorithm is initialized with a single plaintext sequence containing the byte value P_1^* for position $r = 1$ and with the estimated likelihood $\lambda_{P_1^*} = 1$. Now, using the above described method for extending a plaintext byte sequence and the corresponding likelihood estimate, the attack algorithm iterates over the range of considered positions as follows. For each position r, and for all possible values μ_{r+1} of the plaintext byte at position $r+1$, the extension with μ_{r+1} of each of the sequences from the previous iteration is considered, and, for each of the possible values of μ_{r+1}, the algorithm stores the "most likely" extended sequence having μ_{r+1} as the last byte value (that is, it stores the extended sequence which maximises the likelihood estimate expressed in equation (3)). When the attack algorithm reaches the last position, it simply returns the sequence with the highest likelihood estimate.

Note that this process is guaranteed to find the plaintext byte sequence with the highest likelihood estimate computed according to equation (3). However, we emphasise that this expression yields *only* an approximation to the actual plaintext likelihood, being based on the twin assumptions that plaintext bytes are independently and uniformly distributed and that keystream bytes have no dependencies beyond those in adjacent bytes as expressed in the double-byte distributions.

A full description of the attack algorithm is given in Algorithm 4 (on page 22). Note that the algorithm can easily be extended to work for the case where the plaintext byte at the initial position is unknown. In particular, by exploiting the single-byte biases, the likelihoods of all possible values of the initial plaintext byte can be estimated, and subsequently used as a starting point for the algorithm. Of course, the algorithm need not start at position $r = 1$ either.

Notice that the algorithm involves heavy nesting of loops, particularly in phase 2b, where for each position r we perform a computation over all possible values for the candidate plaintext byte pair (μ_{r-1}, μ_r), each such computation itself involving a sum over 2^{32} pairwise products of real numbers arising from the triple summation over $\overline{\texttt{TSC}}$, k_1 and k_2. Thus a direct implementation of this algorithm would require on the order of 2^{48} additions and products *per position*! This would be inconvenient, to say the least. For this reason, and because our double-byte, per-$\overline{\texttt{TSC}}$ keystream distributions are not particularly accurate (being based only on 2^{30} keystreams each), we would in preference use aggregated versions of the algorithm. Specifically, building on our experience in the single-byte case, we may consider a version of the algorithm that works with $\texttt{TSC}_0$-aggregated distributions and only works on a per-$\texttt{TSC}_1$ basis. It is not hard to see how to modify Algorithm 4 to operate in this way, saving a factor of 2^8 in its computational cost. The algorithm could be further modified to use fully aggregated distributions, saving another factor of 2^8 in computational cost, but now effectively ignoring any $\overline{\texttt{TSC}}$-related structure in the keystream distributions.

We have performed a very limited validation of our double-byte attack in its fully aggregated form. A complete evaluation of the algorithm and a comparison of its performance with the single-byte Algorithm 3 is deferred to the full version of the paper. We make one observation at this stage, however. Algorithm 4 makes use of ratios of probability expressions of the form $\beta_r^{(\mu_r,\mu_{r+1})}/\alpha_r^{(\mu_r)}$, where the numerator is a double-byte probability and the numerator is a single-byte probability. If the significant biases in the former probabilities actually arise from products of single-byte biases for adjacent positions, then such expressions can be simplified to just single-byte probability terms of the form $\alpha_{r+1}^{(\mu_{r+1})}$, in effect reducing our double-byte attack to our single-byte attack. Such behaviour can be expected in early byte positions, where single-byte biases are very large. Thus we do not expect our double-byte attack in Algorithm 4 to significantly out-perform our single-byte attack in the early positions. On the other hand, in regions where single-byte biases become smaller and fewer in number but double-byte biases still persist (as seems to be the case in later positions), then Algorithm 4 may be expected to perform better than our single-byte attack. Indeed, Algorithm 4 should be able to smoothly interpolate between regions where single-byte biases dominate and regions where they do not.

6 MPPE

6.1 Computing Keystream Distributions for MPPE Keys

We also computed the RC4 keystream distributions for the first 256 keystream bytes using MPPE keys having the structure described in Section 2.3. More specifically, we generated random 8-byte random keys $\mathtt{K} = (\mathtt{K}_0, \ldots, \mathtt{K}_7)$ and then overwrote key bytes according to the MPPE specification for the 40-bit and 56-bit cases, while in the 128-bit case, we generated random 16-byte keys. We used more than 2^{39} keys in each case, with all computations being performed on our local computing facilities. Figure 5 compares the distributions obtained for random 128-bit RC4 keys (as used in 128-bit MPPE and in TLS) with those for 40-bit and 56-bit MPPE keys. As can be seen, the process of fixing certain key bytes to constant values produces many additional, strong biases in the corresponding keystreams.

6.2 Attack Simulation Results

We used the MPPE keystream distributions to simulate plaintext recovery attacks using the algorithm of [1], equivalent to the fully aggregated version of Algorithm 3. The results are depicted in Figure 6. As expected, the additional structure in RC4 keys introduced by MPPE in the 40-bit and 56-bit cases significantly aids plaintext recovery, with 40-bit keys leading to the highest success rate for a given number of ciphertexts. We also experimented with random 64-bit keys, finding success rates very close to the random 128-bit case. This indicates that it is not the reduction in key-size that makes the difference in MPPE, but rather the introduction of fixed key bytes.

7 Conclusions

In this paper, we have explored the use of cloud computing facilities to perform large-scale computations in support of the cryptanalysis of WPA/TKIP. We expended 63 virtual-core-years of computational effort at a cost of US$43k to carry out two computations, one involving 2^{48} keystreams to estimate per-$\overline{\mathtt{TSC}}$ single-byte distributions, the other involving 2^{46} keystreams to estimate per-$\overline{\mathtt{TSC}}$ double-byte distributions. The total amount of computation was roughly one-twentieth of that used in the sieving stage for the factorisation of RSA-768[10]. The problems of developing efficient code for, and then managing, these computations were not insignificant but ultimately surmountable. This suggests that commercial cloud services can be used as a platform for this kind of work, instead of relying on owned infrastructure. Certainly, running 2^{13} hyper-threaded cores in parallel was an exhilarating, if expensive, way to explore the limits of commercial cloud computing capabilities.

The value of our keystream distribution computations for WPA/TKIP is aptly illustrated in Figure 4, which shows the marked improvement in success rate that accrues from moving from single-byte keystream distribution estimates based on 2^{24} keystreams per $\overline{\mathtt{TSC}}$ to 2^{32} keystreams per $\overline{\mathtt{TSC}}$. Our computations of RC4 keystream distributions in WPA/TKIP and MPPE also provide experimental data that may be useful in making hypotheses about keystream biases, and which may in turn lead to a better theoretical understanding of the operation of RC4 in these applications. Certainly, having an explanation for the long-lived $\mathtt{TSC}_1$-specific single-byte biases that we observed experimentally would be very welcome. A similar project would investigate the effect of fixing key bytes in RC4 keys, and apply the results to provide a theoretical explanation for the observed biases in MPPE keystreams.

Our attack on WPA/TKIP based on double-byte biases requires further investigation: the time and budget available for this project has limited our experimentation with it and reduced our investment in its fine-tuning. Given the dominance of single-byte biases in early portions of the RC4 keystreams for WPA/TKIP, we expect this algorithm to come into its own when targeting repeated plaintext that is located later in WPA/TKIP frames (e.g. after position 256). Moreover, it provides a mechanism for smoothly transitioning attacks from the regime where single-byte biases dominate to the regime where these biases are no longer apparent but where double-byte biases are still present. It remains to investigate whether other forms of bias (such as the "ABSAB" biases from [11]) can be effectively integrated into a more general Bayesian approach, and how much impact this might have on overall attack performance.

Acknowledgements

The research of the authors was conducted in part while all authors were at Royal Holloway, University of London. The research was supported by an EPSRC Leadership Fellowship, EP/H005455/1 as well as a grant from the UK government. We thank Martin Albrecht, Jon Hart and Adrian Thomas at RHUL for their assistance with sourcing, building and maintaining our local computing infrastructure and for help in managing AWS. We thank Strombenzin for its generous donation of computing cycles. We thank the UK government for financing our adventures with Amazon's cloud computing infrastructure, and Mark Rowlands

[10] Estimated at 1500 core-years for a single core 2.2 GHz AMD Opteron processor with 2GB RAM in [9].

at Amazon Web Services for his assistance in maxing out the AWS US West data centre.

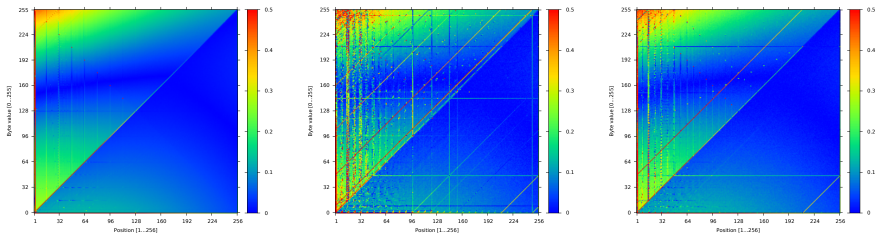
References

1. N. J. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt. On the security of RC4 in TLS. In *USENIX Security*. USENIX Association, 2013. https://www.usenix.org/conference/usenixsecurity13/security-rc4-tls.
2. S. R. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of RC4. In S. Vaudenay and A. M. Youssef, editors, *Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.
3. S. R. Fluhrer and D. McGrew. Statistical analysis of the alleged RC4 keystream generator. In B. Schneier, editor, *FSE*, volume 1978 of *Lecture Notes in Computer Science*, pages 19–30. Springer, 2000.
4. D. Hulton and M. Marlinspike. Divide and conquer: Cracking MS-CHAPv2 with a 100% success rate, 2012. https://www.cloudcracker.com/blog/2012/07/29/cracking-ms-chap-v2/.
5. IEEE 802.11. Wireless LAN medium access control (MAC) and physical layer (PHY) specification, 1997.
6. IEEE 802.11i. Wireless LAN medium access control (MAC) and physical layer (PHY) specification: Amendment 6: Medium access control (MAC) security enhancements, 2004.
7. T. Isobe, T. Ohigashi, Y. Watanabe, and M. Morii. Full plaintext recovery attack on broadcast RC4. In S. Moriai, editor, *FSE*, volume 8424 of *Lecture Notes in Computer Science*, pages 179–202. Springer, 2013. ISBN 978-3-662-43932-6.
8. K. Jaganathan, L. Zhu, and J. Brezak. The RC4-HMAC Kerberos Encryption Types Used by Microsoft Windows. RFC 4757 (Informational), Dec. 2006. http://www.ietf.org/rfc/rfc4757.txt.
9. T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. J. J. te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In T. Rabin, editor, *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350. Springer, 2010. ISBN 978-3-642-14622-0. URL http://dx.doi.org/10.1007/978-3-642-14623-7_18.
10. S. Maitra, G. Paul, and S. Sen Gupta. Attack on broadcast RC4 revisited. In A. Joux, editor, *FSE*, volume 6733 of *Lecture Notes in Computer Science*, pages 199–217. Springer, 2011.
11. I. Mantin. Predicting and distinguishing attacks on RC4 keystream generator. In R. Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 491–506. Springer, 2005.
12. I. Mantin and A. Shamir. A practical attack on broadcast RC4. In M. Matsui, editor, *FSE*, volume 2355 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2001.
13. T. Ohigashi, T. Isobe, Y. Watanabe, and M. Morii. How to recover any byte of plaintext on RC4. In T. Lange, K. Lauter, and P. Lisonek, editors, *Selected Areas in Cryptography*, volume 8282 of *Lecture Notes in Computer Science*, pages 155–173. Springer, 2013. ISBN 978-3-662-43413-0.

14. G. Pall and G. Zorn. Microsoft Point-To-Point Encryption (MPPE) Protocol. RFC 3078 (Informational), Mar. 2001. http://www.ietf.org/rfc/rfc3078.txt.
15. K. G. Paterson, B. Poettering, and J. C. N. Schuldt. Plaintext recovery attacks against WPA/TKIP. In *FSE 2014*, Lecture Notes in Computer Science. Springer, to appear.
16. S. Sarkar, S. Sen Gupta, G. Paul, and S. Maitra. Proving TLS-attack related open biases of RC4. Cryptology ePrint Archive, 2013/502. https://eprint.iacr.org/2013/502.
17. B. Schneier and Mudge. Cryptanalysis of Microsoft's Point-to-Point Tunneling Protocol (PPTP). https://www.schneier.com/paper-pptp.pdf.
18. S. Sen Gupta, S. Maitra, W. Meier, G. Paul, and S. Sarkar. Dependence in IV-related bytes of RC4 key enhances vulnerabilities in WPA. In *FSE 2014*, Lecture Notes in Computer Science. Springer, to appear.
19. S. Sen Gupta, S. Maitra, G. Paul, and S. Sarkar. (Non-) random sequences from (non-) random permutations – analysis of RC4 stream cipher. *Journal of Cryptology*, 27(1):67–108, 2014.
20. P. Sepehrdad, S. Vaudenay, and M. Vuagnoux. Statistical attack on RC4 – distinguishing WPA. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 343–363. Springer, 2011.
21. E. Tews and M. Beck. Practical attacks against WEP and WPA. In D. A. Basin, S. Capkun, and W. Lee, editors, *WISEC*, pages 79–86. ACM, 2009.
22. M. Vanhoef and F. Piessens. Practical verification of WPA-TKIP vulnerabilities. In K. Chen, Q. Xie, W. Qiu, N. Li, and W.-G. Tzeng, editors, *ASIACCS*, pages 427–436. ACM, 2013.
23. G. Zorn. Deriving Keys for use with Microsoft Point-to-Point Encryption (MPPE). RFC 3079 (Informational), Mar. 2001. http://www.ietf.org/rfc/rfc3079.txt.

Algorithm 4: Double-byte bias attack

```
input  : $C$ – balanced vector of $2^{16} \cdot S$ encryptions of fixed plaintext $P$
         ($C_{\overline{\mathtt{TSC}},j,r}$ denotes $r$-th byte of $j$-th encryption of $P$ for TSC-value $\overline{\mathtt{TSC}}$)
         $L$ – length of $P$ in bytes
         $m_1$ and $m_L$ – known first and last byte of $P$
         $\{p_{\overline{\mathtt{TSC}},r,k}\}_{\overline{\mathtt{TSC}} \in \mathcal{T}sc\mathcal{S}p,\, 1 \le r \le L,\, k \in \mathcal{B}yte}$ – single-byte key distribution
         $\{\tilde{p}_{\overline{\mathtt{TSC}},r,k_1,k_2}\}_{\overline{\mathtt{TSC}} \in \mathcal{T}sc\mathcal{S}p,\, 1 \le r < L,\, k_1,k_2 \in \mathcal{B}yte}$ – double-byte key distribution
output: estimate $P^*$ for plaintext $P$
begin
    $N_{\overline{\mathtt{TSC}},r,k} \leftarrow 0$    for all $\overline{\mathtt{TSC}} \in \mathcal{T}sc\mathcal{S}p$, $1 \le r \le L$, $k \in \mathcal{B}yte$
    $\tilde{N}_{\overline{\mathtt{TSC}},r,k_1,k_2} \leftarrow 0$    for all $\overline{\mathtt{TSC}} \in \mathcal{T}sc\mathcal{S}p$, $1 \le r < L$, $k_1, k_2 \in \mathcal{B}yte$
    initialise mappings $Q, Q' \colon \mathcal{B}yte \to \mathcal{B}yte^* \times \mathbb{R}$
    // Phase 1    (count occurrences of keystream bytes and byte pairs)
    for each $\overline{\mathtt{TSC}} \in \mathcal{T}sc\mathcal{S}p$ do
        for $j = 1$ to $S$ do
            for $r = 1$ to $L-1$ do
                $N_{\overline{\mathtt{TSC}},r,C_{\overline{\mathtt{TSC}},j,r}} \leftarrow N_{\overline{\mathtt{TSC}},r,C_{\overline{\mathtt{TSC}},j,r}} + 1$
                $\tilde{N}_{\overline{\mathtt{TSC}},r,C_{\overline{\mathtt{TSC}},j,r},C_{\overline{\mathtt{TSC}},j,r+1}} \leftarrow \tilde{N}_{\overline{\mathtt{TSC}},r,C_{\overline{\mathtt{TSC}},j,r},C_{\overline{\mathtt{TSC}},j,r+1}} + 1$
    // Phase 2a    (derive likelihoods for plaintext byte at position 2)
    for $\mu_2 = \mathtt{0x00}$ to $\mathtt{0xFF}$ do
        $\lambda_{m_1 \| \mu_2} \leftarrow + \sum_{\overline{\mathtt{TSC}} \in \mathcal{T}sc\mathcal{S}p} \sum_{k_1,k_2 \in \mathcal{B}yte} \tilde{N}_{\overline{\mathtt{TSC}},1,k_1 \oplus m_1,k_2 \oplus \mu_2} \log \tilde{p}_{\overline{\mathtt{TSC}},1,k_1,k_2}$
                      $- \sum_{\overline{\mathtt{TSC}} \in \mathcal{T}sc\mathcal{S}p} \sum_{k \in \mathcal{B}yte} N_{\overline{\mathtt{TSC}},1,k \oplus m_1} \log p_{\overline{\mathtt{TSC}},1,k}$
        $Q[\mu_2] \leftarrow (\mu_2, \lambda_{m_1 \| \mu_2})$
    // Phase 2b    (derive likelihoods for plaintext bytes at positions $3 \ldots (L-1)$)
    for $r = 3$ to $L-1$ do
        for $\mu_r = \mathtt{0x00}$ to $\mathtt{0xFF}$ do
            $L^* \leftarrow -\infty$
            for $\mu_{r-1} = \mathtt{0x00}$ to $\mathtt{0xFF}$ do
                parse $Q[\mu_{r-1}]$ as $(P', \lambda_{P'})$
                $\lambda_{P' \| \mu_r} \leftarrow \lambda_{P'}$
                    $+ \sum_{\overline{\mathtt{TSC}} \in \mathcal{T}sc\mathcal{S}p} \sum_{k_1,k_2 \in \mathcal{B}yte} \tilde{N}_{\overline{\mathtt{TSC}},r-1,k_1 \oplus \mu_{r-1},k_2 \oplus \mu_r} \log \tilde{p}_{\overline{\mathtt{TSC}},r-1,k_1,k_2}$
                    $- \sum_{\overline{\mathtt{TSC}} \in \mathcal{T}sc\mathcal{S}p} \sum_{k \in \mathcal{B}yte} N_{\overline{\mathtt{TSC}},r-1,k \oplus \mu_{r-1}} \log p_{\overline{\mathtt{TSC}},r-1,k}$
                if $\lambda_{P' \| \mu_r} > L^*$ then
                    $(P^*, L^*) \leftarrow (P', \lambda_{P' \| \mu_r})$
            $Q'[\mu_r] \leftarrow (P^* \parallel \mu_r, L^*)$
        $Q \leftarrow Q'$
    // Phase 3    (pick most likely plaintext out of candidate set)
    $L^* \leftarrow -\infty$
    for $\mu_{L-1} = \mathtt{0x00}$ to $\mathtt{0xFF}$ do
        parse $Q[\mu_{L-1}]$ as $(P', \lambda_{P'})$
        $\lambda_{P' \| m_L} \leftarrow \lambda_{P'}$
            $+ \sum_{\overline{\mathtt{TSC}} \in \mathcal{T}sc\mathcal{S}p} \sum_{k_1,k_2 \in \mathcal{B}yte} \tilde{N}_{\overline{\mathtt{TSC}},L-1,k_1 \oplus \mu_{L-1},k_2 \oplus m_L} \log \tilde{p}_{\overline{\mathtt{TSC}},L-1,k_1,k_2}$
            $- \sum_{\overline{\mathtt{TSC}} \in \mathcal{T}sc\mathcal{S}p} \sum_{k \in \mathcal{B}yte} N_{\overline{\mathtt{TSC}},L-1,k \oplus \mu_{L-1}} \log p_{\overline{\mathtt{TSC}},L-1,k}$
        if $\lambda_{P' \| m_L} > L^*$ then
            $(P^*, L^*) \leftarrow (P', \lambda_{P' \| m_L})$
    return $m_1 \parallel P^* \parallel m_L$
```



(a) 128-bit MPPE keys (b) 40-bit MPPE keys (c) 56-bit MPPE keys

Fig. 5. Pictorial representation of biases in RC4 keystreams for 128-bit, 40-bit and 56-bit MPPE keys, for different positions (x-axis) and byte values (y-axis). For each position we encode the bias in the keystream for the (position,value) combination as a colour; in each case, the colouring scheme encodes the absolute biases, i.e., the absolute difference between the occurring probabilities and the (expected) probability 1/256, scaled up by a factor of 2^{16}, capped to a maximum of 0.5.

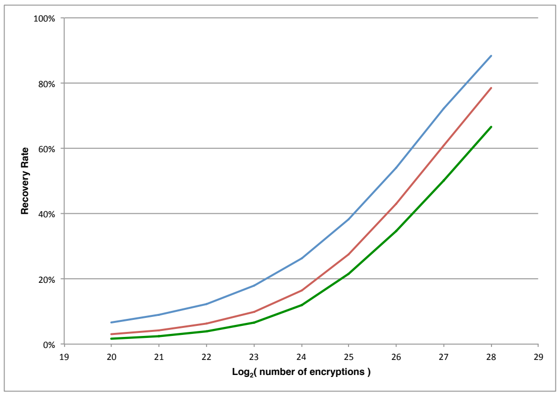


Fig. 6. Average success rates of single-byte plaintext recovery attacks against MPPE using 40-bit keys (blue), 56-bit keys (red), and 128-bit keys (green) over positions 1 to 256. The success rates are based on 256 experiments.