Cryptanalysis of the “Kindle” Cipher

Alex Biryukov, Gaëtan Leurent, and Arnab Roy

University of Luxembourg
{alex.biryukov,gaetan.leurent,arnab.roy}@uni.lu

Abstract. In this paper we study a 128-bit-key cipher called PC1 which is used as part of the DRM system of the Amazon Kindle e-book reader. This is the first academic cryptanalysis of this cipher and it shows that PC1 is a very weak stream cipher, and can be practically broken in a known-plaintext and even in a ciphertext-only scenario.

A hash function based on this cipher has also been proposed and is implemented in the binary editor WinHex. We show that this hash function is also vulnerable to a practical attack, which can produce meaningful collisions or second pre-images.

Keywords: Cryptanalysis, Stream cipher, Hash function, Pukall Cipher, PC1, PSCHF, MobiPocket, Amazon Kindle, E-book.

1 Introduction

In this paper we study the stream cipher PC1, a 128-bit key cipher designed by Pukall in 1991. The cipher was first described in a Usenet post [7] and implementations of the cipher can be found on the designer’s website [9][1]. The PC1 cipher is a part of the DRM system of the MOBI e-book format, which is used in the Amazon Kindle and in MobiPocket (a popular free e-book reader which supports a variety of platforms). This fact makes this cipher into one of the most widely deployed ciphers in the world with millions of users holding devices with this algorithm inside. This cipher is also used in a hashing mode by WinHex [10], an hexadecimal editor used for data recovery and forensics.

So far, no proper security analysis of PC1 is available in the academic literature. Thus it is interesting to study the security of PC1 due to its widespread use and because it offers a nice challenge to cryptanalyst. Our results show practical attacks on the PC1 stream cipher. First, we show a known plaintext attack which recovers the key in a few minutes with a few hundred kilobytes of encrypted text (one small book). Second, we show a ciphertext-only attack, using a secret text encrypted under one thousand different keys. We can recover the plaintext in less than one hour, and we can then use the first attack to extract the keys if needed. Additionally, we show that the hashing mode is extremely weak, by building a simple second-preimage attack with complexity 2^{24}, and a more advanced attack using meaningful messages with a similar complexity. Our results are summarized in Table 1.

[1] Implementations of PC1 can also be found in various DRM removal tools.

2 Description of PC1

PC1 can be described as a self-synchronizing stream cipher, with a feedback from the plaintext to the internal state. The cipher uses 16-bit integers, and simple arithmetic operations: addition, multiplication and bitwise exclusive or (xor). The round function produces one byte of keystream, and the plaintext is encrypted byte by byte.

The internal state of the cipher can be described as a 16-bit integer s, and an 8-bit integer π which is just the xor-sum of all the previous plaintext bytes ($\pi^t = \oplus_{i=0}^{t-1} p^i$). The round function PC1Round takes as input the 128 bit key k and the state (s, π), and will produce one byte of keystream and a new value of the state s.

In this paper we use the following notations:

p	Plaintext	$+$	Addition modulo 2^{16}
c	Ciphertext	$\times$	Multiplication modulo 2^{16}
σ	Keystream	$\oplus$	Boolean exclusive or (xor)
k_i	16-bit sub-keys: $k = k_0 \| k_1 \dots \| k_7$		
x^t	The value of x after t iterations of PC1Round		
$x[i]$	Bit i of x. We use $x[i–j]$ or $x[i, \dots, j]$ to denote bits i to j.		
$f(x) = x \times 20021 + 1$		$g_i(x) = (x + i) \times 20021$	
$h(x) = x \times 346$		$\mathsf{fold}(x) = (x \gg 8) \oplus x \pmod{2^8}$	

A schematic description of PC1 is shown in Figure 1, and pseudo-code is given in Figure 2. We can divide the PC1 in two parts, as shown in the figure:

– The first part is independent of the state s and takes only the key and the state π as input. We denote this part as KF (key function), and it produces two outputs: $w = \mathsf{KF}_1(\pi, k)$ is a set of 8 values used by the second part, and $\sigma_k = \mathsf{KF}_2(\pi, k)$ is used to create the keystream.
– The second part updates the state s from the previous value of s, and the value of the w's. We denote this part as SF (state function), and it produces two outputs: $\mathsf{SF}_1(s, w)$ is the new state s, and $\sigma_s = \mathsf{SF}_2(s, w)$ is used to create the keystream.

A high-level representation of PC1 using these functions is given in Figure 3. An important property of PC1 is that the only operations in KF and SF are modular additions, modular multiplications, and bitwise xors (the f, g_i, and h functions only use modular additions and multiplication). Therefore, KF and SF are T-functions [5]: we can compute the i least significant bits of the outputs by knowing only the i least significant bits of the inputs. The only operation that is not a T-function in the PC1 design is the fold from 16 bits to 8 bits at the end.

Note that our description of PC1 does not follow exactly available code: we use an equivalent description in order to make the state more explicit. In particular, we put g_0 at the end of the round in order to only have a 16-bit state s, while the reference code keeps two variables and computes g_0 and the subsequent sum at the beginning of the round. We also use an explicit π state instead of modifying the key in place.

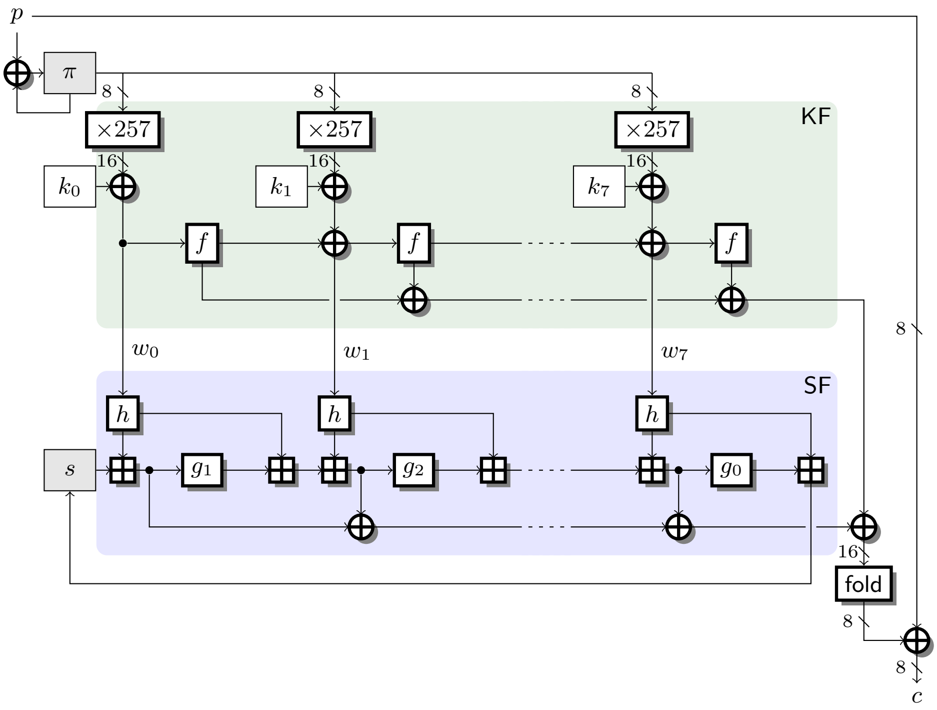


Fig. 1. The PC1 stream cipher

2.1 Use in the Mobipocket e-Book Format and in the Kindle

A notable use of the PC1 stream cipher is in the MOBI e-book format [6]. This format allows optional encryption with PC1; this feature is used to build the DRM scheme of MobiPocket and of the Amazon Kindle. An encrypted e-book is composed of plaintext meta-data and several encrypted text segments. Each segment contains 4 kB of text with HTML-like markup, and is optionally compressed with LZ77. This implementation of LZ77 keeps most characters as-is in the compressed stream, and use non-ASCII characters to encode length-distance pairs. In practice there are still many repetitions in the compressed stream, most of them coming from the formatting tags.

It is a well established fact the DRM system of both MobiPocket and the Amazon Kindle are based on PC1. As a verification, we downloaded several e-books from the Amazon store and they all followed this format.

Each text segment in a given e-book is encrypted with the same key, and the PC1 stream cipher does not use any IV. Thanks to the plaintext feedback the corresponding keystreams will not all be the same. Nonetheless the lack of IV implies a significant weakness: the first byte of keystream will be the same for all encrypted segments, so we can recover the first character of each segment by knowing the first character of the file. Moreover we can detect when two segments share a common prefix.

```
function PC1Round($k$,$\pi$,$s$)
    $k \to k_0, k_1, \ldots, k_7$
    $\sigma \leftarrow 0$
    $w \leftarrow 0$
    for $0 \leq i < 8$ do
        $w \leftarrow w \oplus k_i \oplus (\pi \times 257)$
        $x \leftarrow h(w)$                              ▷ $h(x) = x \times 346$
        $w \leftarrow f(w)$                              ▷ $f(x) = x \times 20021 + 1$
        $s \leftarrow s + x$
        $\sigma \leftarrow \sigma \oplus w \oplus s$
        $s \leftarrow g_{i+1 \bmod 8}(s) + x$            ▷ $g_i(x) = (x + i) \times 20021$
    $\sigma \leftarrow \mathsf{fold}(\sigma)$            ▷ $\mathsf{fold}(x) = (x \gg 8) \oplus x \pmod{2^8}$
    return $(\sigma, s)$

function PC1Encrypt($k$, $p$)
    $\pi \leftarrow 0$;    $s \leftarrow 0$
    for all $p^t$ do
        $(\sigma, s) \leftarrow$ PC1Round($k, \pi, s$)
        $c^t \leftarrow p^t \oplus \sigma$
        $\pi \leftarrow \pi \oplus p^t$
    return $c$

function PC1Decrypt($k$, $c$)
    $\pi \leftarrow 0$;    $s \leftarrow 0$
    for all $c^t$ do
        $(\sigma, s) \leftarrow$ PC1Round($k, \pi, s$)
        $p^t \leftarrow c^t \oplus \sigma$
        $\pi \leftarrow \pi \oplus p^t$
    return $p$
```

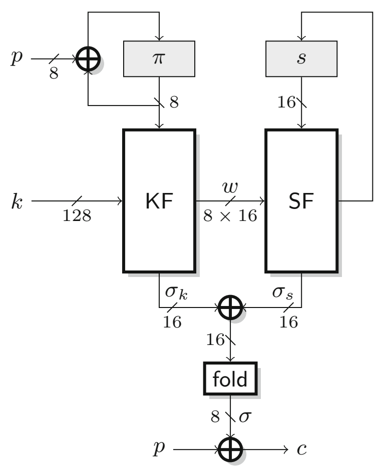
Fig. 2. Pseudo-code of the PC1 stream cipher

Attacks on the DRM Scheme. Like all DRM schemes, this system is bound to fail because the key has to be present in the device and can be extracted by the user. Indeed this DRM scheme has been reverse engineered, and software is available to decrypt the e-books in order to read them with other devices [3].

In this paper, we do not look at the DRM part of the system, but we target the stream cipher from a cryptanalysis point of view.

3 Previous Analysis

Two simple attacks on PC1 have already been described. Our new attacks will exploit some of the same properties — which we rediscovered — and expand on those ideas in order to build practical key-recovery attacks.



$$
\begin{aligned}
\pi^{t+1} &= \pi^t \oplus p^t \\
w^t &= \mathsf{KF}_1(\pi^t, k) \\
s^{t+1} &= \mathsf{SF}_1(s^t, w^t) \\
\sigma^t &= \mathsf{fold}(\mathsf{KF}_2(\pi^t, k) \oplus \mathsf{SF}_2(s^t, w^t)) \\
c^t &= p^t \oplus \sigma^t
\end{aligned}
$$

Fig. 3. Overview of PC1. Grey boxes represent memory registers.

Table 1. Summary of the attacks on the PC1 stream cipher, and on the PSCHF hash function

Attacks on PC1		Complexity	Data	Reference
Distinguisher	Chosen plaintext	2^{16}	2^{16}	[1]
Key recovery	Known plaintext	2^{72}	2^{4}	[2]
Key recovery	Known plaintext	2^{31}	2^{20}	Section 5
Key recovery	Ciphertext only, 2^{10} unrelated keys	2^{35}	$2^{17}.2^{10}$	Section 6
Attacks on PSCHF		**Complexity**		**Reference**
Second preimage	with meaningful messages	2^{24}		Section 7

3.1 Key Guessing

As described above, most of the computations of PC1 can be seen as a T-function. Therefore, if we guess the low 9 bits of each sub-key, we can compute the low 9 bits of KF and SF in a known plaintext attack. This gives one bit of the keystream after the folding, and we can discard wrong guesses. We can then guess the remaining key bits, and the full attack has a complexity of 2^{72}. This was described in a Usenet post by Hellström [2].

3.2 State Collisions

Our description clearly shows that the internal state of the stream cipher is very small: 8 bits in π that do not depend of the key, and 16 bits in s. Therefore we expect that there will be collisions in the state quickly. This was first reported by

Hellström on Usenet in [1], where he described a chosen-plaintext distinguisher: given two messages $x_0 \| y$ and $x_1 \| y$ such that x_0 and x_1 have the same xor sum, the encryption of y will be the same in both messages with probability 2^{-16}.

A more efficient distinguisher can be built using the birthday paradox. We consider 2^8 different prefixes x_i with a fixed xor sum, and a fixed suffix y. When encrypting the messages $x_i \| y$, we expected that two of them will show the same encryption of y when the state s collides after encrypting x_i and x_j.

4 Properties of PC1

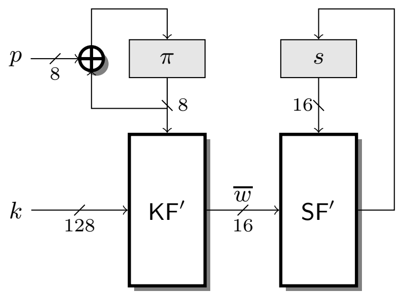
Before describing our attacks, we study some useful properties of the design of PC1.

4.1 Simplified State Update

First, we can see that the SF_1 function only uses modular additions and modular multiplications by constants. Therefore, the state update can be written as a degree 1 polynomial (the full coefficients are given in Appendix A):

$$s^{t+1} = \mathsf{SF}_1(s^t, w^t) = \sum_{i=0}^{7} \left(a_i \times w_i^t\right) + b \times s^t + c$$

If we integrate the computation of $\overline{w} = \sum_{i=0}^{7} a_i \times w_i$ inside KF, we only have to transmit 16 bits between KF and SF for the s update loop. This results in the simplified state update of Figure 4 (we denote the resulting functions by KF' and SF').



$$\begin{aligned} \overline{w}^t &= \mathsf{KF}'(\pi^t, k) \\ s^{t+1} &= \mathsf{SF}'(s^t, \overline{w}^t) \\ &= \overline{w}^t + b \times s^t + c \end{aligned}$$

Fig. 4. Simplified PC1 state update

In the following, we denote $\mathsf{KF}'(x, k)$ as $\overline{w}_x$ for a given key k. In particular, we have $\overline{w}^t = \overline{w}_{\pi^t}$. One can see that knowing the values of $\overline{w}_x$ for all possible x is sufficient to compute the state update without knowing the key itself. Equivalently, we can see KF' as a key-dependent 8×16 bit S-Box.

4.2 Diffusion

All operations in KF and SF are T-function so the only diffusion is from the low bits to the high bits. Moreover, several bits actually cancel out and only affect output bits at higher indexes. We can learn how the key affects the state update by looking at the coefficients a_i. We notice an interesting property of the least significant bits of the coefficients:

$$\forall i, \qquad a_i \equiv 4 \bmod 8 \qquad b \equiv -1 \bmod 8 \qquad c \equiv 4 \bmod 8$$

Therefore, if bits 0 to i of the key and plaintext are known, we can compute bits 0 to i of w, and bits 0 to $i+2$ of s. More precisely, we can write

$$\sum_{i=0}^{7} a_i \times w_i = 4 \times \sum_{i=0}^{7} w_i + 8 \times \sum_{i=0}^{7} a'_i \times w_i \qquad \text{with } a_i = 8a'_i + 4$$

$$s^{t+1} \equiv 4 \times \sum_{i=0}^{7} w_i - s^t + 4 \pmod 8$$

We also have $\sum_{i=0}^{7} w_i \equiv \bigoplus_{i=1,3,5,7} k_i \pmod 2$ and $s^0 = 0$, which leads to:

$$s^t \equiv 4 \times t \times \left(1 \oplus \bigoplus_{i=1,3,5,7} k_i\right) \pmod 8$$

In particular, this shows that $s^t \equiv 0 \bmod 4$ (the two least significant bits of s^t are always zero), and we will always have $s^{t+2} = s^t \bmod 8$. Collisions between s^t and $s^{t'}$ will be more likely if t and t' have the same parity; more generally, collisions are more likely when $t - t'$ is a multiple of large power of two, but the exact relations are difficult to extract.

We can also see that keys with $\bigoplus_{i=1,3,5,7} k_i = 1$ will lead to more frequent collisions, because 3 bits of s are fixed to zero. This defines a class of key keys with regard to collision based attacks. More generally we can define classes of increasingly weak keys for which more low bits of the state are fixed.

5 Collision-Based Known Plaintext Attack

As mentioned above, collisions in the internal state are relatively likely due to the small state size. We show how to use such collisions in an efficient key recovery attack.

First, let us see how we can detect state collisions. If a collision happens between steps t and t' (*i.e.* $s^t = s^{t'}$ and $\pi^t = \pi^{t'}$) this will result in $\sigma^t = \sigma^{t'}$ and $s^{t+1} = s^{t'+1}$. Additionally, if the plaintexts at positions t and t' match, we will have $c^t = c^{t'}$ and $\pi^{t+1} = \pi^{t'+1}$, which will in turn give $\sigma^{t+1} = \sigma^{t'+1}$ and $s^{t+2} = s^{t'+2}$. Furthermore, if several bytes of the plaintext match, this will give a match in several keystream bytes because the state transitions will be the same. More formally, we have

$$\begin{cases} s^t = s^{t'} \\ \pi^t = \pi^{t'} \\ p^{t,\dots,t+u-1} = p^{t',\dots,t'+u-1} \end{cases} \implies \begin{cases} s^{t+1,\dots,t+u+1} = s^{t'+1,\dots,t'+u+1} \\ \sigma^{t+1,\dots,t+u+1} = \sigma^{t'+1,\dots,t'+u+1} \\ c^{t+1,\dots,t+u} = c^{t'+1,\dots,t'+u} \end{cases}$$

A u-byte match in the plaintext results in a u-byte match in the ciphertext, plus one byte in the keystream, provided that the state (s, π) also matches.

In order to exploit this in a key-recovery attack, we look for matches in the plaintext and ciphertext, and we assume that they correspond to state collisions. If we get enough colliding bytes we will have few false positives, and we will learn that $s^t = s^{t'}$ for some values of t and t'. This is very valuable because the computation of s from k is a T-function. We can then recover the key bit by bit: if we guess the least significant bits of k we can compute the least significant bits of s and verify that they collide.

We now study some internal details of the cipher in order to speed-up this key recovery and to make it more practical.

5.1 Detecting State Collisions

We look for pairs of positions (t, t') with:

$$\pi^t = \pi^{t'} \quad \textbf{and} \quad p^{t,\dots,t+u-1} = p^{t',\dots,t'+u-1} \tag{A}$$

$$\sigma^{t+1,\dots,t+u+1} = \sigma^{t'+1,\dots,t'+u+1} \tag{B}$$

Condition (A) is a $u+1$-byte condition depending only on the plaintext (we have $\pi^t = \bigoplus_{i=0}^{t-1} p^i$), while condition (B) is a $u+1$-byte condition depending also on the ciphertext. In our attack we use $u = 2$: we have a 3-byte filtering to detect the two-byte event $s^t = s^{t'}$, and we expect few false positives.

However, due to the structure of the cipher, the probability of having (B) is bigger than $2^{-8(u+1)}$, even when $s^t \neq s^{t'}$. First, the most significant bit of s does not affect σ, because its effect is cancelled by the structure of additions and xors. More generally, if we have $s^t \equiv s^{t'} \bmod 2^i$ (*i.e.* an i-bit match in s), then $i+1$ bits of σ will match, and this implies $\Pr(\mathrm{B}) \geq 2^{-(16-i-1)(u+1)}$. For instance, with $u = 2$, we have $\Pr(\mathrm{B}) \geq 2^{-2}$ when $s^t \equiv s^{t'} \bmod 2^{14}$, which would generate many false positives.

In our implementation of the attack, when we detect (B), we only assume that this correspond to $s^t \equiv s^{t'} \bmod 2^{10}$. Experimentally, the probability of detecting (B) with $u = 2$ is around 2^{-15} for random keys and random s states, versus 2^{-24} if $s^t \not\equiv s^{t'} \bmod 2^{10}$. Therefore we have $\Pr\left[s^t \not\equiv s^{t'} \bmod 2^{10} \;\middle|\; (\mathrm{B})\right] \approx 2^{-24}/2^{-15} = 2^{-9}$, and we expect to have no false positives when we use a dozen collisions.

Since we only assume that 10 bits of s are colliding, we can only use these collisions to recover the low 8 bits of the subkeys. For the upper 8 bits of the key we use the output stream σ. If we know $k_j[0–7]$ and we guess $k_j[8]$, we can compute $\sigma_k[0–8] \oplus \sigma_s[0–8]$ before the fold, and one bit of σ after the fold. We can verify our guess by comparing this to the known least significant bit of $c \oplus p$.

Note that most text documents have a relatively low entropy, therefore condition (A) will be satisfied with probability significantly higher than $2^{-8(u+1)}$. In practice, with a sample book of 183 kB (after LZ77 compression), we have 120959 pairs of positions satisfying (A), and for a random encryption key we usually detect between six and one hundred collisions.

5.2 Key Recovery

The basic approach to use those collisions in a key recovery attack is to guess the key bits one by one, and to compute the state in order to exclude wrong guesses.

In a known plaintext attack if we guess $k_j[0–i]$ for all j then we can compute $w_j[0–i]$ for all j, and also $s[0, \dots, i+2]$. We can verify a guess by comparing s^t and $s^{t'}$ up to the bit $i+2$.

However, this essentially requires us to perform a trial encryption of the full text to test each key guess. To build a significantly more efficient attack we consider how s is updated from the key. As explained in Section 4, we have $s^{t+1} = \overline{w}_{\pi^t} + b \times s^t + c$, where the $\overline{w}_x$ can be computed from the key. For a given plaintext p, we can compute π^t at each step, and each s^t can be written as a linear combination of the $\overline{w}_x$:

$$s^t = R^t(\overline{w}_0, \dots, \overline{w}_{255})$$

with the following relations:

$$R^t = \overline{w}_{\pi^t} + b \times R^{t-1} + c \qquad\qquad R^0 = 0$$

We can compute the coefficients of each R^t from the known plaintext, and every state collision we detect can be translated to an equality $R^t = R^{t'}$. For each guess of the least significant bits of the key, checking those equalities only requires to compute the 256 values $\overline{w}_x$, and to evaluate linear combinations with 256 terms.

Moreover, we can look for sparse relations, so that we don't have to evaluate all the 256 values $\overline{w}_x$. First, note that we also have implicit relations due to the structure of KF': we know that KF is a T-function, and the coefficients a_i used to compute $\overline{w}$ are all multiple of 4; this gives $\overline{w}_x \equiv \overline{w}_y \mod 2^{i+2}$ for all x and y with $x \equiv y \mod 2^i$. We use MAGMA to compute the vector space generated by the collision relations, and we compute the quotient of this space by the implicit relations. The basis of the quotient contains relatively sparse relations, and we find very sparse ones (with only one or two terms) when we restrict the equations to $k[0–i]$ for a small i, by working in the ring $\mathbf{Z}/2^i\mathbf{Z}$.

Using these relations, a key trial now costs less than 256 evaluations of the round function. In practice this gives a speedup of about 100 times over the staightforward approach.

5.3 Dealing with Independent Message Segments

As mentionned in section 2.1, MOBI e-books are divided into several segments. Each of these segments is encrypted with the same key starting with the initial state $(\pi, s) = (0, 0)$. The segments are too short to find collisions *inside* a given segment, but we can use collisions *between* two different segments just as easily.

Let's assume we detect a collision in the state s, π after enciphering the plaintext p_1 from the initial state and after enciphering the plaintext p_2 from the initial state. We can verify a guess of the least significant bits of the key by enciphering p_1 and p_2 starting from the initial state, and verifying that the least significant bits of the states match.

5.4 Complexity of the Attack

It is difficult to give the precise complexity of the attack because it depends on the number of collisions found, and how much filtering they give. We did some experiments to measure the actual complexity by enciphering a fixed book with random keys, as reported by Figure 5. We found that when we have at least 9 collisions, the median complexity is less than 2^{23} key trial, which take about one minute. The attack is still practical with as low as six collisions, but in this case we have to try around 2^{30} key candidates, which takes a few hours with one core of a typical PC.

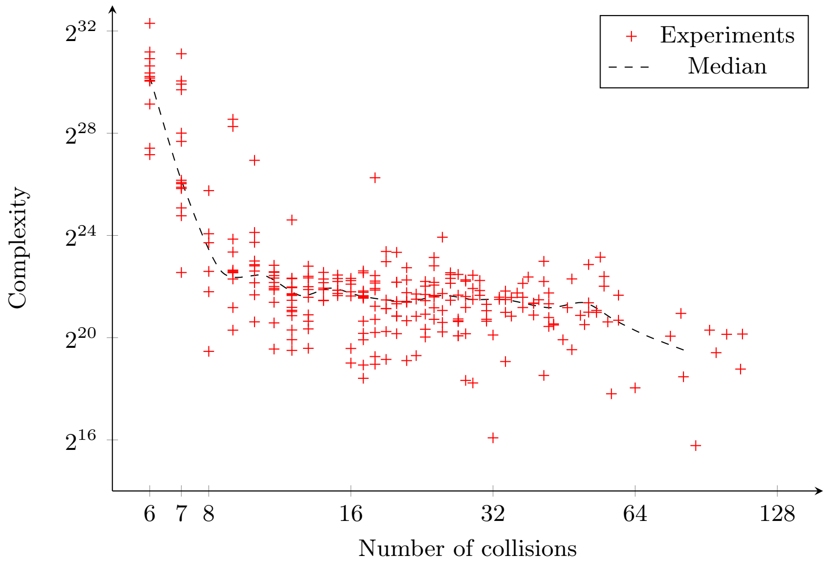


Fig. 5. Experimental results with an e-book of size 336kB (after LZ77 compression). Complexity is shown as the number of key trials.

With this sample e-book, the vast majority of keys result in more than 9 collisions, and often a lot more. This results in a complexity of less that 2^{23} key trials. In general, this will be the case if the known plaintext is sufficiently long (several hundred kilobytes, or a megabyte). Since each key trial costs less than 2^8 evaluation of the round function, the attack will cost less than 2^{31} evaluations of the round functions. In practice, it can be done in less than one minute.

6 Ciphertext Only Attack Using Many Unknown Keys

We now describe a ciphertext-only attack, assuming that the attacker has access to several encryptions of the same text under many different and unrelated keys. If a DRM scheme is based on PC1, this would be a collusion attack where several users buy a copy of a protected work and they share the encrypted data. This attack is based on the observation that the keystream generated by a random key at two differents positions $\sigma_1 = PC1(k, s_1, \pi_1)$ and $\sigma_2 = PC1(k, s_2, \pi_2)$ are biased when $\pi_1 = \pi_2$.

For each plaintext position t, we build a vector C^t with the corresponding ciphertext under each available key. If we consider a pair of positions t and t' the vectors C^t and $C^{t'}$ will be correlated if $\pi^t = \pi^{t'}$. If we manage to detect this correlations efficiently, we can "color" the text positions with 256 colors corresponding to the values of π^t. Then we only have to recover the actual value of π^t corresponding to each color. We can use some known part of the text, the low entropy of the human language, or some extra information recovered when detecting the bias.

Bias in $\boldsymbol{\sigma}$[0]. Let us first study the bias between the C^t vectors. The main bias is in the least significant bit, and is present when $\pi^t[0\text{–}6] = \pi^{t'}[0\text{–}6]$. Let us consider a given plaintext p encrypted under a random key, and two positions t, t' with this property. Because of cancellation effects in the structure of KF and SF, $\pi[7]$ does not affect bits 0–8 of $\sigma_k = \mathsf{KF}_2(\pi, k)$ and $\sigma_l = \mathsf{SF}_2(s, w)$. Moreover, with some probability we have $s^t[0\text{–}8] = s^{t'}[0\text{–}8]$. If this if the case, we will have $\sigma_l^t[0\text{–}8] = \sigma_l^{t'}[0\text{–}8]$, and $\sigma^t[0] = \sigma^{t'}[0]$ after the fold.

This results in a bias in $c^t[0] \oplus c^{t'}[0]$:

$$\begin{aligned}
&\Pr\left[c^t[0] = c^{t'}[0] \;\middle|\; \pi^t[0\text{–}6] = \pi^{t'}[0\text{–}6]\right]\\
&= \Pr\left[p^t[0] \oplus \sigma^t[0] = p^{t'}[0] \oplus \sigma^{t'}[0] \;\middle|\; \pi^t[0\text{–}6] = \pi^{t'}[0\text{–}6]\right]\\
&= \Pr\left[\sigma^t[0] \oplus \sigma^{t'}[0] = p^t[0] \oplus p^{t'}[0] \;\middle|\; \pi^t[0\text{–}6] = \pi^{t'}[0\text{–}6]\right]\\
&\approx 1/2 \pm \Pr\left[s^t[0\text{–}8] = s^{t'}[0\text{–}8]\right]
\end{aligned}$$

The bias is positive if $p^t[0] = p^{t'}[0]$ and negative otherwise. As noted in Section 4.2, two bits of s are fixed to zero, which results in $\Pr\left[s^t[0\text{–}8] = s^{t'}[0\text{–}8]\right] \geq 2^{-7}$. We even have three fixed bits if $t' \equiv t \pmod 2$, which give a stronger bias of 2^{-6}. Moreover, we can get even stronger biases for classes of weak keys, and when using positions such that $t - t'$ is a multiple of a larger power of 2.

Bias with More Bits of $\boldsymbol{\sigma}$. There are similar biases with more outputs bits when $\pi^t = \pi^{t'}$. For instance, we have $s^t[0\text{–}9] = s^{t'}[0\text{–}9]$, with some probability. This implies $\sigma_l^t[0\text{–}9] = \sigma_l^{t'}[0\text{–}9]$, and $\sigma^t[0\text{–}1] = \sigma^{t'}[0\text{–}1]$ after the fold. This results in a bias in $c^t[0\text{–}1] \oplus c^{t'}[0\text{–}1]$:

$$\Pr\left[c^t[0\text{–}1] \oplus c^{t'}[0\text{–}1] = p^t[0\text{–}1] \oplus p^{t'}[0\text{–}1] \;\middle|\; \pi^t = \pi^{t'}\right] \approx \frac{1}{4} + \Pr\left[s^t[0\text{–}9] = s^{t'}[0\text{–}9]\right]$$

6.1 Clustering

These biases are quite strong and most colors can be recovered if we have access to a fixed text encrypted under 2^{20} different keys. In order to reduce the number of keys needed, we use a more elaborate algorithm.

First, we work on sets of positions with the same remainder modulo 8: $\mathcal{T}_i = \{t \mid t \equiv i \bmod 8\}$. Positions in the same set will show a stronger bias on $\sigma[0]$: $\Pr\left[s^t[0\text{–}8] = s^{t'}[0\text{–}8] \mid t \equiv t' \pmod 8\right]$ is about 2^{-6} for strong keys, but it can be as high as 2^{-4} for weaker keys (one key in 8 is weak). This allows to detect some relations with only one thousand keys. Each relation also gives the value of $p^t[0] \oplus p^{t'}[0]$ from the sign of the bias.

Then we use a clustering algorithm to detect positions wich share the same color. Initially, we assign a different color to each position, and we merge pairs of colors when we detect a significant bias (we use a priority queue to start with the strongest bias, and we recompute the bias as the clusters grow). When comparing clusters with more than one position, we effectively have a larger sample size, and we can detect weaker biases. Note that we need to correct the signs of the biases using the values of $p^t[0] \oplus p^{t'}[0]$ that we recover when merging colors.

The first phase of the algorithm stops when have identified 128 large colors. We assume that these correspond to the 128 values of $\pi[0\text{–}6]$, which generate the bias in $\sigma[0]$. We then remove false positives from each cluster by verifying that each position is strongly correlated to the rest of the cluster, and we go through all the unassigned positions and assign them to the cluster with the stongest correlation (again, we use a priority queue to start with the strongest bias).

At this point each $\mathcal{T}_i$ has been partitioned in 128 colors, corresponding to the value of $\pi[0\text{–}6]$. We then match the colors of different $\mathcal{T}_i$'s by choosing the strongest correlation (we first merge $\mathcal{T}_i$ and $\mathcal{T}_{i+4}$ because this allows bigger biases, then $\mathcal{T}_i$ and $\mathcal{T}_{i+2}$, and finally $\mathcal{T}_i$ and $\mathcal{T}_{i+1}$).

Finally, we have to split each color: we have the value of $\pi[0\text{–}6]$, but we want to recover the full value of $\pi[0\text{–}7]$. We use the bias on $\sigma[0\text{–}1]$ to detect when two positions correspond to the same $\pi[0\text{–}7]$ (note that we already know the value of $p^t[0] \oplus p^{t'}[0]$). For every color, we first pick two random points to create the new colors, and we assign the remaining points to the closest group. We repeat this with new random choices until the same partition is found three times.

The full attack is given as pseudo-code in Algorithms 1 and 2.

Algorithm 1. Pseudo-code of the clustering algorithm

```
for all $i$ do                                    ▷ Initially, assign a different color to every position
    Color[$i$] ← $i$

for $0 \le x < 8$ do                                                              ▷ For each $\mathcal{T}_x$
    repeat                                          ▷ Merge colors with the strongest correlation
        for all $i, j \equiv x \pmod 8$, s.t. Color[$i$] $\neq$ Color[$j$] do
            $b$ ←ComputeBias( Get(Color[$i$]), Get(Color[$j$]) )
            if $b > b_{max}$ then
                $b_{max} \leftarrow b$; $c_i \leftarrow$ Color[$i$]; $c_j \leftarrow$ Color[$j$]
        for all $k \in$ Get($c_i$) do
            Color[$k$] ← $c_j$
    until 128 large colors have been identified                        ▷ Store in MainColor[$x$]
    for all $i \equiv x \pmod 8$ do                                          ▷ Remove false positives
        if ComputeBias( $\{i\}$, Get(Color[$i$]) $\setminus \{i\}$ ) $> \epsilon$ then
            Color[$i$] ← NewColor()

    for all $i \equiv x \pmod 8$ do                 ▷ Assign remaining points to the closest color
        for $0 \le j < 128$ do
            $b$ ←ComputeBias( $\{i\}$, Get(MainColor[$x$][$j$]) )
            if $b > b_{max}$ then
                $b_{max} \leftarrow b$; $c \leftarrow$ MainColor[$x$][$j$]
        Color[$i$] ← $c$

Merge(0,4); Merge(1,5); Merge(2,6); Merge(3,7);
Merge(0,2); Merge(1,3); Merge(0,1);                         ▷ Merge colors from different $\mathcal{T}_x$
for $0 \le i < 128$ do                                  ▷ Split colors from $\pi$[0–6] to $\pi$[0–7]
    Split(Get(MainColor[0][$i$]))
return Color
```

6.2 Experiments

We performed experiments with a sample text of a few kilobytes that we encrypted with PC1 under 2^{10} different keys. In this setting, our clustering algorithm can recover the colors in half an hour with a desktop PC. To associate the correct π value to each color, we can use the fact the MOBI format encrypts each chunk independently, and that the first character of a book is always a tag opening character "<". This allows to recover the first byte of each segment, and to identify the colors. In the end, we can decipher the full text with only a few errors. From that point, we can use the known-plaintext attack of Section 5 to recover the keys, and to produce a clean plaintext.

One of the most expensive steps of the attack is to compute the bias between each pair of positions in the plaintext. In our implementation, we use 2^{17} bytes of text, divided in 8 sets $\mathcal{T}_i$ of size 2^{14}. Therefore we have to compute 8×2^{27} biases, and each computation requires 2^{10} bit operations, or 2^5 word operations. Therefore the complexity of the attack is about 2^{35} word operations.

Algorithm 2. Functions used by the clustering algorithm

```
function SPLIT($\mathcal{S}$)                          ▷ Split color from $\pi[0–6]$ to $\pi[0–7]$
    repeat
        $a, b \leftarrow$ RANDOM($\mathcal{S}$); $\mathcal{A} \leftarrow \{a\}$; $\mathcal{B} \leftarrow \{b\}$
        for all $i \in \mathcal{S}$ do
            if COMPUTEBIAS2($\{i\}, \mathcal{A}$) > COMPUTEBIAS2($\{i\}, \mathcal{B}$) then
                $\mathcal{A} \leftarrow \mathcal{A} \cup \{i\}$
            else
                $\mathcal{B} \leftarrow \mathcal{B} \cup \{i\}$
    until the same partition $\mathcal{A}, \mathcal{B}$ is found three times
    $c \leftarrow$ NEWCOLOR()
    for all $i \in \mathcal{A}$ do
        Color[$i$] $\leftarrow c$

function MERGE($x$, $y$)                        ▷ Merge colors from different $\mathcal{T}_x$
    for $0 \le j < 128$ do
        for $0 \le i < 128$ do
            $b \leftarrow$ COMPUTEBIAS( GET(MainColor[$x$][$i$]), GET(MainColor[$y$][$j$]) )
            if $b > b_{max}$ then
                $b_{max} \leftarrow b$; $c \leftarrow$ MainColor[$x$][$i$]
        for all $k \in$ GET(MainColor[$y$][$j$]) do
            Color[$k$] $\leftarrow c$

function GET($c$)                               ▷ Returns the set of positions currently in color $c$
    return $\{i \mid \text{Color}[i] = c\}$

function COMPUTEBIAS($\mathcal{S}$, $\mathcal{S}'$)        ▷ Evaluates the bias in $\sigma[0]$ between $\mathcal{S}$ and $\mathcal{S}'$

function COMPUTEBIAS2($\mathcal{S}$, $\mathcal{S}'$)       ▷ Evaluates the bias in $\sigma[0–1]$ between $\mathcal{S}$ and $\mathcal{S}'$
```

7 PSCHF: A Hash Function Based on PC1

A hash function based on PC1 has also been proposed by Pukall [8], and it is used in the WinHex hexadecimal editor to check the integrity of a file. The PSCHF hash function operates in two steps:

- First the message is encrypted with PC1 using a fixed key k_h, and the encrypted message is cut into chunks of 256 bits which are xor-ed together to produce an intermediate 256-bit value h.
- Second, a *finalization* function is computed from the final value of the state (s, π), h and the message length $\alpha \pmod{32}$.

The pseudo-code for the hash function is given in Figure 6.

7.1 Second Preimage Attack

To conclude our analysis, we describe a second preimage attack against this hash function. We ignore the finalization, and target the values h, s, π, α after the main loop.

```
$h[0,\dots,31] \leftarrow 0$
$s \leftarrow 0$, $\pi \leftarrow 0$
$\alpha \leftarrow 0$
▷ The first loop reads the input message
for all $p^t$ do
    $(\sigma, s) \leftarrow \mathrm{PC1R{\scriptsize OUND}}(k_h, \pi, s)$
    $h[\alpha] \leftarrow h[\alpha] \oplus \sigma \oplus p^t$
    $\pi \leftarrow \pi \oplus p^t$
    $\alpha \leftarrow \alpha + 1 \bmod 32$
▷ The second loop is a finalization whose input are $h$, $s$, $\pi$, and $\alpha$
for $0 \le j < 10 \times (\ell + 1)$ do
    $(\sigma, s) \leftarrow \mathrm{PC1R{\scriptsize OUND}}(k_h, \pi, s)$
    $\pi \leftarrow \pi \oplus h[\alpha]$
    $h[\alpha] \leftarrow \sigma$
    $\alpha \leftarrow \alpha + 1 \bmod 32$
return $h$
```

Fig. 6. Pseudo-code of the PSCHF hash function. The key k_h is a fixed constant, and ℓ is used to compute the number of blank rounds in the finalization. WinHex uses $\ell = 10$ and $k_h =$ `0xF6C72495179F3F03C6DEF156F82A8538`.

We use $E(M, s, \pi)$ to denote the encryption of a message block M with the key k_h, starting from state (s, π). The intermediate value h can be written as:

$$h = E(M_0, \pi^0, s^0) \oplus E(M_1, \pi^{32}, s^{32}) \oplus \cdots \oplus E(M_l, \pi^t, s^t),$$

where the s^t, π^t values are implicitly computed by the previous E calls. The message blocks are 32-byte long, but the last one might be incomplete.

We can easily build a second preimage attack due to the small internal state of the cipher. We consider a given message $\overline{M}$, and the corresponding target state $\overline{h}, \overline{s}, \overline{\pi}, \overline{\alpha}$ before the finalization function. First, let us assume that the length of $\overline{M}$ is a multiple of 32 bytes, *i.e.* $\overline{\alpha} = 0$. We consider a two-block message $M = M_0, M_1$, and we want to reach the pre-specified value $\overline{h}$:

$$h = E(M_0, \pi^0, s^0) \oplus E(M_1, \pi^{32}, s^{32}) = \overline{h},$$

or equivalently:

$$E(M_1, \pi^{32}, s^{32}) = \overline{h} \oplus E(M_0, \pi^0, s^0).$$

We can find solutions by picking M_0 randomly and just compute M_1 by decrypting $\overline{h} \oplus E(M_0, \pi^0, s^0)$, starting from the state (π^{32}, s^{32}) reached after encrypting M_0. Note that we have $\alpha = 0$ because we use a message of length 64 bytes. However, we also need to reach the pre-specified internal state *i.e.* $s^{64} = \overline{s}, \pi^{64} = \overline{\pi}$. For a random choice of M_0 this should be satisfied with probability 2^{-21} (the probability is higher than 2^{-24} because at least three bits of s are fixed to zero).

If the length of the given message $\overline{M}$ is not a multiple of 32 we can still mount a similar attack. We use a message M made of three parts: M_0 of length α, M_1 of length $32-\alpha$, and M_2 of length α. A preimage has to satisfy:

$$h = \big(E(M_0, \pi^0, s^0) \| E(M_1, \pi^\alpha, s^\alpha)\big) \oplus \big(E(M_2, \pi^{32}, s^{32}) \| 0^{32-\alpha}\big) = \overline{h}$$
$$\big(E(M_2, \pi^{32}, s^{32}) \| E(M_1, \pi^\alpha, s^\alpha)\big) = \big(E(M_0, \pi^0, s^0) \| 0^{32-\alpha}\big) \oplus \overline{h}$$

Like in the previous case, we can choose a random M_0, and obtain M_1 by decrypting $\overline{h}[\alpha+1, \ldots, 31]$ (starting from the state (π^α, s^α) found after M_0) and M_2 by decrypting $E(M_0, \pi^0, s^0) \oplus \overline{h}[0, \ldots, \alpha]$ (starting from the state (π^{32}, s^{32}) found after computing M_1). At the end, we have $\pi^{32+\alpha} = \overline{\pi}$ and $s^{32+\alpha} = \overline{s}$ with probability 2^{-21}.

We can also use the attack with a chosen prefix. Given a target message $\overline{M}$ and a chosen prefix $\overline{N}$, we can build M such that $H(\overline{N} \| M) = H(\overline{M})$.

This attack has been verified and examples of second preimage are given in Table 2. These examples are preimages of the empty message; they can be used as a prefix to any chosen message $\overline{M}$ and will provide a message $P \| \overline{M}$ with the same hash value. In this setting, it is also possible to build a meaningful message: if the message block M_0 is meaningful and goes to the state $s = 0, \pi = 0$, then the decryption of M_0 will give $M_1 = M_0$ and the full message is meaningful.

Table 2. Examples of second preimage of the empty message. We use the same key as in WinHex: k_h = `0xF6C72495179F3F03C6DEF156F82A8538`.

Random Message															
D5	06	35	27	03	5C	71	E0	F6	D8	49	9B	C9	ED	95	B2
FE	38	1E	A0	A5	26	1A	80	91	F8	53	2E	EF	5D	54	C4
FC	8B	F0	09	D2	5C	5A	36	08	D6	41	F8	34	F5	50	5D
96	F6	C5	30	56	4A	9C	0D	E2	DA	29	FD	4C	4A	F0	62
Meaningful Message (hex)															
2A	20	20	44	4F	20	6E	4F	74	20	52	45	41	44	20	74
68	69	73	20	6D	65	73	73	61	67	65	20	21	20	20	0A
2A	20	20	44	4F	20	6E	4F	74	20	52	45	41	44	20	74
68	69	73	20	6D	65	73	73	61	67	65	20	21	20	20	0A
Meaningful Message (ASCII)															
`*␣␣DO␣nOt␣READ␣this␣message␣!␣␣`															
`*␣␣DO␣nOt␣READ␣this␣message␣!␣␣`															
$H(M) = H(\varnothing)$															
51	DE	77	DF	24	04	D0	37	18	DE	7C	53	9E	8A	62	75
FA	48	B0	3C	E3	C1	5F	31	4D	58	F8	D8	FF	3B	19	8D

7.2 Meaningful Preimages

More generally, we can build arbitrary preimage where we control most of the text, using Joux's multi-collision structure [4], and a linearization technique.

First, we consider 2^8 meaningful blocks with the same xor-sum, and we compute the state after encrypting them. We expect that two block $m_{0,0}$ and $m_{0,1}$ will lead to the same state s^{32}, π^{32}. We repeat this from the state s^{32}, π^{32} to find two messages $m_{1,0}$ and $m_{1,1}$ that lead to the same state s^{64}, π^{64}, and we build a multi-collision structure with 256 pairs $m_{i,0}, m_{i,1}$ iteratively. This structure contains 2^{256} different messages all leading to the same state s^{8192}, π^{8192}. Each step needs 2^8 calls to PC1Round, so the full structure will be built for a cost of 2^{16}.

We then add a final block m_{256} of size $\overline{\alpha}$ and whose xor-sum is $\overline{\pi} \oplus \pi^{8192}$, in order to connect the state s^{8192}, π^{8192} to the target state $\overline{s}, \overline{\pi}$. This require 2^{16} trials. We now have 2^{256} messages all going to the correct $\overline{s}, \overline{\pi}$ and $\overline{\alpha}$, and we will select one that goes to the correct $\overline{h}$ using a linearization technique.

Let us define $c_{i,x} = E(m_{i,x}, \pi^{32i}, s^{32i})$ and $c_{256} = E(m_{256}, \pi^{8192}, s^{8192})$. We can then express h as a function of 256 unknown x_i's:

$$\begin{aligned} h &= c_{0,x_0} \oplus c_{1,x_1} \oplus \cdots \oplus c_{255,x_{255}} \oplus c_{256} \\ &= c_{256} \oplus \bigoplus_{i=0}^{255} c_{i,x_i} \\ &= c_{256} \oplus \bigoplus_{i=0}^{255} c_{i,0} \oplus \bigoplus_{i=0}^{255} x_i \cdot (c_{i,0} \oplus c_{i,1}) \\ &= C \oplus X \cdot D, \end{aligned}$$

where $C = c_{256} \oplus \bigoplus_{i=0}^{255} c_0$, D is a matrix whose row are the $c_{i,0} \oplus c_{i,1}$, and X is a row vector of the x_i's. We can then solve $\overline{h} = C \oplus X \cdot D$ using linear algebra, and we find a message that is a preimage of $\overline{M}$. This technique will produce meaningful second preimages of length around 256 block, *i.e.* 8 kilobytes, with a complexity of 2^{24}.

8 Conclusion

In this work, the study the cipher PC1, which is used in the Amazon Kindle as part of the DRM scheme. Our analysis target the cipher itself, and not the full DRM scheme. We show devastating attacks against the PC1 cipher, and the PSCHF hash function: a known-plaintext key-recovery attack, a ciphertext only attack using a thousand unrelated keys, and a meaningful second-preimage attack on the hash function. All these attacks are practical and have been implemented. While trying to make our attacks more efficient, we have used cryptanalytic techniques which could be of independent interest.

Our attack scenarios are practical: if a DRM scheme is based on PC1, colluding users can recover the plaintext from a thousand ciphertexts encrypted with different keys. This analysis shows that PC1 is very weak and probably made its way into popular products due to the lack of academic cryptanalysis, which we provide in this paper. However, the practical impact on existing DRM schemes is limited, because there are already easy ways to circumvent them.

The main problem in the design of PC1 is the very small internal state, which allows attacks based on internal collisions. Additionnaly, our attacks exploit the fact that several components of PC1 are T-functions, *i.e.* the diffusion is only from the low bits to the high bits.

Acknowledgment. Gaëtan Leurent is supported by the AFR grant PDR-10-022 of the FNR Luxembourg.

References

1. Hellström, H.: Re: Good stream cipher (other than ARCFOUR). Usenet post on sci.crypt (January 18, 2002) Message id: S8K18.14572$l93.3141016@newsb.telia.net
2. Hellström, H.: Re: stream cipher mode. Usenet post on sci.crypt (February 3, 2002) Message id: 3C5CA721.9080905@streamsec.se
3. i♡cabbages: Circumventing Kindle For PC DRM (updated). Blog entry (December 20, 2009) `http://i-u2665-cabbages.blogspot.com/2009/12/circumventing-kindle-for-pc-drm.html`
4. Joux, A.: Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 306–316. Springer, Heidelberg (2004)
5. Klimov, A., Shamir, A.: A New Class of Invertible Mappings. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 470–483. Springer, Heidelberg (2003)
6. MobileRead: MobileRead Wiki — MOBI (2012), `http://wiki.mobileread.com/w/index.php?title=MOBI&oldid=30301` (accessed May 14, 2012)
7. Pukall, A.: crypto algorithme PC1. Usenet post on fr.misc.cryptologie (October 3, 1997) Message id: 01bcd098$56267aa0$LocalHost@jeushtlk
8. Pukall, A.: Description of the PSCHF hash function. Usenet post on sci.crypt (June 9, 1997) Message id: 01bc74aa$ae412ae0$1aa54fc2@dmcwnjdz
9. Pukall, A.: The PC1 Encryption Algorithm – Very High Security with 128 or 256-bit keys (2004), `http://membres.multimania.fr/pc1/`
10. WinHex: WinHex webpage, `http://www.x-ways.net/winhex/`

A Details of the State Update Polynomial

$$\begin{aligned} s^{t+1} &= \mathsf{SF}_1(s^t, w^t) \\ &= 8460 \times w_0^t - 20900 \times w_1^t - 3988 \times w_2^t - 21444 \times w_3^t + 13004 \times w_4^t \\ &\quad - 20196 \times w_5^t + 16428 \times w_6^t - 19204 \times w_7^t - 15007 \times s^t - 29188 \end{aligned}$$