SN Computer Science (2020) 1:226
https://doi.org/10.1007/s42979-020-00221-z

ORIGINAL RESEARCH

PudgyTurtle: Using Keystream to Encode and Encrypt

David A. August[1] · Anne C. Smith[2]

Received: 23 November 2019 / Accepted: 8 June 2020 / Published online: 4 July 2020

Abstract
Stream cipher encryption works by modulo-2 adding plaintext bits to keystream bits, which are in turn produced by successively updating a finite-state machine initialized to a secret starting state. PudgyTurtle is a way to encode the plaintext in a keystream-dependent manner before encryption. Since it can use keystream from any stream cipher, PudgyTurtle functions somewhat like an encryption mode. The process begins by generating successive 4-bit groups of keystream ('nibbles') until one of them matches the current plaintext nibble to within one bit. The number of keystream nibbles required, as well as the nearness of this match, is then encoded into a variable-length codeword. Finally, this codeword is encrypted by modulo-2 addition to an equal amount of keystream. Compared to normal binary-additive stream ciphers, this process is less efficient (i.e., more time is required to generate extra keystream nibbles, and more space is needed for the codewords than for the plaintext). However, with this cost comes a benefit: PudgyTurtle resists time-memory tradeoff attacks better than standard stream encryption.

Keywords Symmetric-key cryptography · Stream ciphers · Time-memory tradeoff · Birthday paradox · Error-correcting codes · Hellman chains

Introduction

Binary-additive stream ciphers (BASC) encrypt by modulo-2 adding each plaintext bit (x_i) to a keystream bit (k_i), thus producing ciphertext $y_i = x_i \oplus k_i$, where $\oplus$ denotes the XOR operation. Because of the self-inverting property of modulo-2 addition, decryption is accomplished in a similar manner: $y_i \oplus k_i = (x_i \oplus k_i) \oplus k_i = x_i \oplus 2k_i = x_i$.

The sequence of bits, $K = k_1, k_2, \ldots$, is called the keystream. K is produced by the *keystream generator* (KSG)—a finite-state machine operating on an n-bit state, S. Since details of the KSG are assumed to be public, its starting-state (S_0) must include a secret key. The starting-state may also incorporate an initialization vector (IV), which need not be secret (e.g., it can be a publicly shared random 'nonce', or can be generated by each communicating party from a counter). Unique IVs allow the same secret key to be used for more than one message.

The KSG works by applying an update function $\pi : \{0,1\}^n \rightarrow \{0,1\}^n$ and an output function $o : \{0,1\}^n \rightarrow \{0,1\}$ to the current n-bit state. Thus, $k_i = o(S_i) = o(\pi(S_{i-1}))$. Functions o and π are designed to make K appear random and unpredictable, which makes it hard to reconstruct any previous state from any sub-sequence of keystream. PudgyTurtle uses the keystream to encode the plaintext into a sequence of variable-length codewords and then encrypts these codewords. This process is cipher-agnostic, in that no constraints are placed on the KSG. Plaintext X is first separated into 4-bit groups ('nibbles'), $X_1, X_2, \ldots$, where $X_i = (x_{4i-3} \| x_{4i-2} \| x_{4i-1} \| x_{4i})$ and $\|$ stands for concatenation. For each X_i, new keystream nibbles are generated until one matches X_i to within a 1-bit tolerance. A *codeword* (C_i) is then created using the number of failed matching attempts, plus some error-correcting information. This codeword is encrypted with a *mask*, M_i—a (possibly non-contiguous) sequence of keystream nibbles of the same length as C_i—thus producing ciphertext

$$Y_i = C_i \oplus M_i$$

David A. August
daugust@mgh.harvard.edu

Anne C. Smith
asmith3142@protonmail.com

[1] MGH-DACCPM / GRJ 444, 55 Fruit St., Boston, MA 02114, USA

[2] McKnight Brain Institute, University of Arizona, Tucson, AZ 85724, USA

Stream-Cipher Cryptanalysis

A major goal of stream-cipher cryptanalysis is to reconstruct any KSG state used during encryption. Once discovered, the generator can then be run forward (and, if π is reversible, backwards) from that state to recreate the keystream and decrypt the ciphertext.

Some stream ciphers are susceptible to direct attacks, which exploit algebraic, key-scheduling, statistical or other structural weaknesses particular to individual crypto-algorithms [11, 21, 36, 38, 39]. If no direct attack is known, then an exhaustive key-search may be attempted. It might seem that, for any KSG with a sufficiently large state-size, this so-called brute-force cryptanalysis would be infeasible. However, this is not always true: cryptanalytic *time-memory-data tradeoffs* (TMDTOs), also called *collision attacks*, are probabilistic methods which take advantage of the Birthday Paradox to speed up an exhaustive search to practical levels (e.g., only $2^{n/2}$ operations instead of 2^n) [16, 26, 29]. These attacks, described more fully in Sect. 5, have proven successful against a variety of stream ciphers [6, 13, 48].

TMDTO attacks against stream ciphers require a sample of *known keystream*, K', obtained by XOR'ing some *known plaintext*, X', to the intercepted ciphertext at the correct offset. PudgyTurtle, however, changes the plaintext–ciphertext relationship into something different than a simple XOR, thus making it harder to obtain K' from X'. On the one hand, this property makes PudgyTurtle less efficient than standard binary-additive stream-ciphers: its codewords take up more space than the original plaintext ('pudgy'), and its encoding procedure needs extra time to match each plaintext nibble to the keystream ('turtle'). On the other hand, these features also increase the difficulty of a collision attack.

Coding and Cryptography

Coding theory has been part of modern cryptology since the work of Shannon in the late 1940's [50]. Hellman extended these ideas for cryptosystem design, commenting on the 'duality' between ciphers and error-correcting codes (ECCs) [25]. Other cryptographers have also investigated various aspects of coding. For example, Bellare and Rogaway explored semantic security and authentication in cryptosystems that include a 'key-less encoding' step (e.g., prepending a counter or IV, and appending a checksum to the message) [7].

Another cryptographic application of coding is 'randomized encryption' [47], in which one ciphertext is chosen randomly from among a family of possibilities. An early instantiation of this idea for asymmetric cryptosystems was by McEliece, who proposed adding random noise after encoding the plaintext with a binary Goppa-type ECC [40]. Here, the private key (G) is the ECC's generator matrix and the public key (G') is obtained by multiplying G by permutation and scrambling matrices. Security arises from the fact that a fast method exists for decoding and error-correction given G, but the generalized linear decoding problem given G' is NP-complete [9]. More recently, similar ideas have appeared in the realm of quantum-resistant cryptography (e.g., lattice-based systems relying on the difficulty of finding the closest vector to one that has been perturbed by noise [45, 46]).

With regard to stream ciphers, several approaches exist for randomized encryption. Kara and Erguler [32, 33] proposed using an ECC to encode the plaintext, which is then encrypted with a 'noisy keystream' (i.e., the modulo-2 sum of the true keystream and random binary noise). The receiver decrypts the ciphertext into a noisy state, then recovers the plaintext by using the ECC. Importantly, the additive noise does not need to be reconstructed by the receiver—unlike a cryptographic key. Another ECC-based approach is based on the 'learning parity with noise' (LPN) problem, analyzed by Fossorier [20]. For example, Imai and Mihaljević [42] proposed an LPN-based system in which external randomness is not just used for additive noise, but also as a basis for homophonic coding (i.e., each plaintext group can be encoded in multiple ways).

PudgyTurtle is similar to these approaches in some ways: its ciphertext is longer than its plaintext; and its encoding changes the plaintext–ciphertext relationship from a simple XOR into something more complex. However, PudgyTurtle also has several important differences. First, unlike some 'encode-then-encipher' ideas [7], PudgyTurtle's encoding is key-dependent, not key-less. Second, PudgyTurtle is deterministic: it is not randomized encryption and does not require an external noise source [32, 33, 40]. Third, PudgyTurtle operates on small (4-bit) plaintext groups: message padding is not required, nor are complex vector and matrix operations. Fourth, PudgyTurtle's codewords do not contain any raw plaintext, unlike other ECC-based systems. Rather, information about the *relative position* of various keystream nibbles is what actually gets encoded. Finally, PudgyTurtle does not suffer from decoding failures, as may (in rare cases) occur with some ECC-based systems [41].

Stream-Cipher Modes

PudgyTurtle is not a cipher itself, but operates alongside existing stream ciphers. In this respect, it resembles an 'encryption mode'. Some stream-cipher modes are intended to add advanced features, like authenticated encryption, by using a BASC as the crypto-primitive [1, 10, 49]. Other stream-cipher modes are designed to resist TMDTO attacks, such as by continuously incorporating the secret key/IV into the KSG's state-update function [2, 3, 43]. The

'FP(1)-mode', used by the LIZARD cipher for instance [24], describes how to re-synchronize the KSG state in a particular key- and IV-dependent manner [23]. In theory, systems like these, which can be analyzed as pseudorandom functions [8], could make TMDTO attacks harder (e.g., by breaking the state-space into a set of mutually exclusive 'keystream equivalence classes', each requiring a separate attack). In practice, security concerns may still exist, due to weak keys, key-scheduling or other issues [15, 18, 19, 24].

Like these stream-cipher modes, PudgyTurtle also attempts to make TMDTO attacks harder. However, rather than altering initialization or re-synchronization procedures, PudgyTurtle instead takes the keystream and uses it in two different ways:

- A binary-additive (codeword $\oplus$ mask) process;
- A non-binary-additive (plaintext-keystream matching) process;

PudgyTurtle can still be compatible with other stream-cipher modes. Whether KSG initialization and re-synchronization happens 'simply' (as in Trivium or A5/1) or by a more complex protocol (as in LIZARD), the keystream produced works equally well for PudgyTurtle.

Outline

After Sect. 2 presents some notation, Sects. 3 and 4 describe the PudgyTurtle algorithm and its output in detail. Section 5 introduces TMDTO attacks, and Sect. 6 discusses some general obstacles to performing such attacks against PudgyTurtle. Section 7 proposes two new TMDTO attacks against PudgyTurtle, and Sect. 8 describes how to quantify their performance. These new attacks are then implemented in Sect. 9, using a toy cipher as the KSG. We show that both attacks are less efficient than two well-known tradeoffs from the literature. Finally, Sect. 10 suggests some potential limitations of PudgyTurtle.

Notation

Hexadecimal values are prefixed by '0x' and binary values have '2' as a subscript. For example, 242 could be written as 0xF2 or 11110010_2.

Hamming weight $h(a)$ is the number of 1's in binary vector a.

Floor function $\lfloor x \rfloor$ is defined as $\lfloor x \rfloor = s$, when $x = s + \epsilon$, assuming that $s \in \{0, 1, 2, \dots\}$ and $\epsilon \in [0, 1)$.

Single vertical bars denote the number of elements in a set (e.g., $|\mathcal{K}|$ is the size of the key-space).

Throughout, X refers to plaintext, K to keystream, and Y to ciphertext, with the following embellishments:

- Lowercase letters with subscripts (x_i, k_i, y_i) represent individual bits.
- Uppercase letters with subscripts denote *groups* of bits: X_i and K_i are (4-bit) nibbles; Y_i is one or more (8-bit) bytes.
- Uppercase letters with a prime ($'$) are assumed to be known to the attacker (e.g., X' is the 'known plaintext' and K' the 'known keystream').
- Uppercase letters with double-subscripts denote individual bytes within a multi-byte symbol. For example, if Y_i is two bytes long, then its first and second bytes are $Y_{i,1}$ and $Y_{i,2}$.
- Uppercase letters with parentheses denote an n-bit segment of a longer sequence, where n is the size of the KSG-state. Depending on context, these segments may be called 'windows', 'prefixes', or 'fragments'. For example, $K(a) = \{k_a, k_{a+1}, k_{a+2}, \dots, k_{a+n-1}\}$ means n bits of keystream starting at bit a.
- $K_{t(i)}$ stands for the keystream nibble that matches plaintext nibble X_i to within 1 bit.
- N_X, N_K, and N_Y represent the *number of symbols* of plaintext, keystream, and ciphertext, whereas L_X, L_K, and L_Y are their *actual lengths* in bits;

PudgyTurtle

This section provides details of PudgyTurtle encryption and decryption. A descriptive overview is offered, followed by an algorithmic explanation. Table 1 can also be consulted as a visual aid.

Overview

The first task is to encode each plaintext nibble X_i into a variable-length codeword, C_i, written as

$$C_i = O_i \parallel F_i \text{ (modulo 32)} \parallel D_i$$

where

- O_i is a variable-length *overflow indicator*, which contains zero or more copies of the special byte 0xFF (see details below);
- $F_i = 0, 1, 2, \dots$ is the *failure-counter*, which counts the number of *un*-successful attempts to match X_i to a keystream nibble. Since F_i is zero-indexed, $F_i = 0$ means that the one keystream nibble had to be generated in order to match X_i, and so on.
- D_i is the *discrepancy-code* (see details below), which describes the mismatch pattern between X_i and $K_{t(i)}$, thereby allowing single-bit error correction by the receiver;

Table 1 PudgyTurtle encryption process

		ASCII 0x48 = "H"		ASCII 0x69 = "i"	
1	Message	ASCII 0x48 = "H"		ASCII 0x69 = "i"	
	Plaintext nibble	4	8	6	9
2	Keystream (hex)	5,3,B,1,4	2,3,0	D,8,5,C,D,2	A,5,7,D
3	Mask (hex)	5 3	2 3	D 8	A 5
4	Failure-counter	0,1,2	0	0,1,2,3	0,1
5	Hamming distance	4,2,0	1	2,2,3,1	3,1
6	Discrepancy-code	4 versus 4 000	8 versus 0 100	6 versus 2 011	9 versus D 011
7	Codeword	00010**000**	00000**100**	00011**011**	00001**011**
8	Mask (binary)	01010011	00100011	11011000	10100101
9	Ciphertext	01000011 0x43	00100111 0x27	11000011 0xC3	10101110 0xAE

Each column illustrates the encryption of one nibble of the plaintext message "Hi" (Row 1, ASCII characters 0x48 and 0x69). Row 2 shows the keystream nibbles. Row 3 depicts the two keystream nibbles set aside for each mask. The failure-counter (Row 4) increments from zero until a keystream nibble matches the plaintext nibble to within a 1-bit tolerance, as quantified by the Hamming distance in Row 5 (e.g., the first Hamming-distance, between K_3 = 0xB = 1011_2 and $X_1 = 4 = 0100_2$, equals 4). When this Hamming distance first becomes ≤ 1, a match occurs, the nearness of which is captured by the discrepancy-code (Row 6). For example, the notation "8 versus 0" above "100" means that $X_2 = 8 = 1000_2$ differs from its matching keystream nibble $K_8 = 0 = 0000_2$ in the most-significant bit, so the discrepancy code is 100_2. Row 7 shows how each codeword is built by concatenating the 5-bit failure counter (normal font) and the 3-bit discrepancy code (boldface). Finally, encryption is accomplished by XOR'ing the mask (shown again in Row 8, as binary) and the codeword, thus producing the ciphertext in Row 9

The PudgyTurtle process begins by saving the first two keystream nibbles ($K_1 \| K_2$) as mask M_1 and setting the first failure-counter F_1 to 0. Then, each new keystream nibble is compared to X_1, starting from K_3. If it differs from X_1 by more than one bit, then F_1 is incremented, a new keystream nibble is generated, and the search continues (i.e., X_1 is then compared to K_4 and so on) until a match is found.

Once the keystream nibble $K_{t(1)}$ that matches X_1 is discovered, the nearness of this match is captured by discrepancy-code $D_1 = \mathrm{d}(X_1, K_{t(1)})$. The general rule for d() is

If $X_i \oplus K_{t(i)}$ is...	then $\mathrm{d}(X_i, K_{t(i)})$ is...
0000_2	$000_2 = 0$
0001_2	$001_2 = 1$
0010_2	$010_2 = 2$
0100_2	$011_2 = 3$
1000_2	$100_2 = 4$

The 8-bit codeword C_1 is then constructed by concatenating F_1 modulo 32 (which is 5 bits) together with D_1 (which is 3 bits). Finally, the ciphertext is produced by encrypting the codeword with the mask: $Y_1 = C_1 \oplus M_1$. This process then repeats for the next plaintext nibble, X_2, starting with $F_2 = 0$ and using keystream beginning at $K_{t(1)+1}$. Table 1 provides a visual example of how a short message is encoded and encrypted via PudgyTurtle.

Overflow Events

This match-encode-encrypt cycle has one caveat: if a failure-counter (say F_1) is ≥ 32, it can no longer be represented by 5 bits, and this *overflow event* triggers a special encoding process: first, an 0xFF byte is pre-pended to C_1; second, mask M_1 is expanded to include the next two available keystream nibbles, which would be K_{35} and K_{36} in this case. That is, $M_1 \leftarrow M_{1,1} \| M_{1,2} = M_{1,1} \ \| \ (K_{35} \| K_{36}) = K_1 \| K_2 \| K_{35} \| K_{36}$. Attempts to match X_1 then continue, starting from keystream nibble K_{37} and $F_1 = 32$. When a match is found, its codeword will be two bytes instead of one:

$$C_1 = C_{1,1} \| C_{1,2} = \mathrm{0xFF} \| (F_1 \bmod 32) \| D_1$$

In the unlikely event that no match occurs even within the next 32 keystream nibbles, this overflow process can be repeated (i.e., both the codeword $C_1 = \mathrm{0xFF} \| \mathrm{0xFF} \| (F_1 \bmod 32) \| D_1$ and mask $M_1 = K_1 \| K_2 \| K_{35} \| K_{36} \| K_{69} \| K_{70}$ would become three bytes long).

The overflow byte 0xFF is made by concatenating the 5-bit failure-counter 31 = 11111_2 together with the 3-bit symbol 111_2. There is no theoretical reason to choose 111_2: any 3-bit discrepancy-code not already in use could also serve (i.e., either 101_2 or 110_2). Practically, however, using 111_2 allows for easy specification of the overflow indicator: if $n_O = \lfloor F_i/32 \rfloor$ is the number of overflow events that occur while encoding X_i, then

$$O_i = \begin{cases} \emptyset & \text{if } n_O = 0 \\ 2^{8n_O} - 1 & \text{if } n_O > 0 \end{cases}$$

In software, ∅ is implemented as an 'empty string'. For example, two overflow events means $O_i = 2^{16} - 1 =$ 0xFFFF.

Because of overflow events, each mask (M_i), codeword (C_i) and ciphertext symbol (Y_i) can be one *or more* bytes. Most of the time, however, there are no overflows, so O_i is the empty string and each of these symbols is just one byte long.

Algorithmic Description

The PudgyTurtle encoding/encryption process can also be conceptualized as an algorithm:

```
1. Initialize
   • Let plaintext nibble index $i = 1$
   • Let keystream nibble index $j = 1$
   • Let the index of the keystream nibble that matched
     the previous plaintext nibble $t(i-1) = t(0) = 0$
   • Let failure counter $F_i = F_1 = 0$
2. FindMatch
   • If $F_i$ mod 32 = 0, then $j \leftarrow j + 2$; endif
   • If $h(X_i \oplus K_j) > 1$, then
         $F_i \leftarrow F_i + 1$
         $j \leftarrow j + 1$
         Go to FindMatch
        else
       $t(i) \leftarrow j$
        endif
3. Encode
   • Make the overflow indicator:
         Let $n_O = \lfloor F_i/32 \rfloor$;
         If $n_O = 0$, then $O_i \leftarrow \emptyset$;
         If $n_O \neq 0$, then $O_i \leftarrow 2^{8n_O} - 1$;
   • Make the discrepancy-code:
         If $X_i = K_{t(i)}$, then $D_i \leftarrow 0$;
         If $X_i \neq K_{t(i)}$, then $D_i \leftarrow 1 + \log_2(X_i \oplus K_{t(i)})$
   • Make the codeword:
       $C_i \leftarrow O_i \| F_i$ (modulo 32)$\| D_i$, where $O_i$ is either
       not present or is a multiple of 8 bits, $F_i$ (mod 32) is
       5 bits, and $D_i$ is 3 bits.
4. MakeMask
   • Let $M_i = \emptyset$ (‘empty string’)
   • For $n = 0$ to $n_O$, do:
       – Let $a = t(i-1) + 1 + 34n$
       – $M_i \leftarrow M_i || (K_a || K_{a+1})$
5. Encrypt
   • $Y_i \leftarrow C_i \oplus M_i$
6. Update
   • $j \leftarrow t(i) + 1$
   • $i \leftarrow i + 1$
   • $F_i \leftarrow 0$
   • Go to FindMatch
```

Decryption

One difference between PudgyTurtle decryption and encryption is that because of overflow events, ciphertext symbols $Y_1, Y_2, \ldots$ may not all be the same length. Put another way, N_Y always equals N_X, but ciphertext length L_Y does not always equal N_Y *bytes*. Thus, decryption requires a separate ‘unmasking’ of individual bytes within each ciphertext symbol.

The *first* byte of Y is unmasked by XOR’ing it with $M_1 = (K_1 \| K_2)$, thereby producing the first byte of the first codeword. If this byte is *not* equal to `0xFF`, then the byte is split into its first 5 bits (failure-counter F_1 modulo 32) and its last 3 bits (discrepancy-code D_1). Next, $F_1 + 1$ new keystream nibbles are generated. The final one of these, $K_{t(1)}$, is the one that matches the original plaintext nibble to within one bit. The plaintext is then recovered by inverting the discrepancy code, as shown below:

If D_1 is...	then X_1 is...
000_2	$K_{t(1)} \oplus 0000_2$
001_2	$K_{t(1)} \oplus 0001_2$
010_2	$K_{t(1)} \oplus 0010_2$
011_2	$K_{t(1)} \oplus 0100_2$
100_2	$K_{t(1)} \oplus 1000_2$

Or more generally,

$$X_i = K_{t(i)} \oplus d^{-1}(D_i) = \begin{cases} K_{t(i)} & \text{if } D_i = 0 \\ K_{t(i)} \oplus 2^{D_i - 1} & \text{if } D_i = 1, 2, 3, \text{or } 4 \end{cases}$$

If, however, unmasking the first byte of Y produces `0xFF`, then an overflow event has occurred (i.e., $Y_1 > 1$ byte long). In this case, 32 keystream nibbles must be generated and discarded, after which the next 2 keystream nibbles $(K_{35} \| K_{36})$ are used to unmask the *second* byte of Y (i.e., $Y_{1,2}$), which is then split into F_1 and D_1 as described above. (In the rare case that $Y_{1,2}$ is also `0xFF`, this overflow process can be repeated.)

Y_2 is decrypted in a similar manner, starting one nibble beyond the current keystream position. That is, the first byte of Y_2 is unmasked by XOR’ing it with $(K_{t(1)+1} \| K_{t(1)+2})$, and—depending upon whether or not the result is `0xFF`—analogous steps are followed. This byte-by-byte unmasking-decoding cycle continues for each of the N_Y ciphertext symbols.

Packet Systems

Some stream ciphers, like E0 for Bluetooth A5/1 for mobile telephony, operate in *packet mode*: the keystream is generated in short segments, and re-synchronized with a new IV or counter after each such packet [23]. For example, A5/1 produces 228 bits of keystream at a time, after which its IV ('frame number') needs to be incremented. With a little extra book-keeping, PudgyTurtle can work with such systems. All that is required is to keep track of the number of available keystream nibbles remaining in the current packet, and then to re-synchronize whenever needed—whether during mask generation, plaintext–keystream matching or an overflow event. The only constraint is that, since PudgyTurtle operates on nibbles, the packet size (in bits) must be a multiple of four.

Statistics

PudgyTurtle's encoding procedure depends upon a random process with an underlying *geometric distribution*: each uniformly distributed keystream nibble either 'succeeds' in matching the current plaintext nibble or 'fails' to match. One success after F failures occurs with probability $g(F,p) = (1-p)^F p$, where $p = 5/16$ describes the five ways a match can happen between two 4-bit symbols (i.e., one exact match plus four 1-bit mismatches).

The mean of this distribution is $1/p = (5/16)^{-1} = 3.2$, which implies that 3.2 keystream nibbles (12.8 keystream bits) on average will be required to match each plaintext nibble.

Overflow events (i.e., $F \geq 32$) occur with probability

$$\begin{aligned} p_O &= 1 - \Pr(F \leq 31) \\ &= 1 - \sum_{j=0}^{31} (1-p)^j \times p \\ &= 6.2047813 \times 10^{-6} \end{aligned}$$

Thus, one overflow event is expected for every $4/p_O \approx$ 644664 bits (80583 bytes).

Ciphertext Length

Because of the probabilistic nature of the plaintext/keystream matching process, the ciphertext length is *not known exactly until after encryption*. For the plaintext, $L_X = 4N_X$ bits. For the ciphertext, however,

$$L_Y = 8(N_X + N_O) \approx 8N_X(1 + p_O)$$

bits, where N_O is the *total* number of overflow events. Thus, the ciphertext includes $(L_Y/8) - N_X$ 'extra' bytes due to the need to encode, on average, $N_O \approx N_X \cdot p_O$ overflow events.

Expansion Factors

The ciphertext expansion factor (CEF) can be written

$$\text{CEF} = \frac{L_Y}{L_X} \approx \frac{8N_X(1 + p_O)}{4N_X} \approx 2$$

The key expansion factor (KEF) is the amount of required keystream as a multiple of the plaintext length. For normal stream-cipher operation, KEF = 1. For PudgyTurtle, KEF ≈ 5.2. This value is obtained by adding 3.2 (the average number of keystream nibbles required to match each plaintext nibble) to 2 (the average number of nibbles used by each mask).

Testing

These predictions were tested using three different plaintext sources: an English-language ASCII document[1] ('English'); a JPEG-formatted digital photograph[2] ('Image'); and a file entirely composed of `0x00` bytes ('Zeros'). A 1280000-byte sample of each plaintext was encrypted using Trivium [14] as the PudgyTurtle KSG, with session key `0x0123456789ABCDEF1234` and initial value `0x6666699999aaaaa55555`. Results are shown in the left half of Table 2. As expected, CEF ≈ 2 and KEF ≈ 5.2.

PudgyTurtle ciphertext should appear random and uniformly distributed no matter what is the underlying statistical structure of the plaintext. This was confirmed using two-sample Kolmogorov–Smirnov tests (right half of Table 2). Specifically, single-byte frequencies were compared between each pair of ciphertexts and also between each ciphertext and a collection of 2560012 uniformly distributed random bytes ('Random'). The non-significant p-values (Column 7) show that the ciphertexts are indistinguishable from one another, and also from random data. At this level of scrutiny, PudgyTurtle does not appear to leak information about its underlying plaintext statistics.

Time Memory Tradeoffs

Here we introduce TMDTO attacks—especially those that target stream-ciphers, and review two in detail. Readers already familiar with these ideas may wish to skip to Sect. 6.

[1] Source: Smith (2002) An Inquiry into the Nature and Causes of the Wealth of Nations. Project Gutenberg, Urbana, Illinois, retrieved December 15, 2018 from www.gutenberg.org/ebooks/19033.

[2] Source: www.webbaviation.co.uk/manchester/towerconstruction-cb18400.jpg.

Table 2 Ciphertext statistics

Plaintext source	Length (bytes)	CEF	KEF	Kolmogorov–Smirnov tests		
				Comparison	KS	*p*-value
English	2560014	2.00	5.2002	Versus random	0.000923	0.26
				Versus image	0.000634	0.68
				Versus zeros	0.000901	0.25
Image	2560017	2.00	5.1996	Versus random	0.000558	0.82
				Versus zeros	0.000485	0.92
Zeros	2560010	2.00	5.1963	Versus random	0.000441	0.96

PudgyTurtle-encryptions of three different 1280000-byte plaintexts: an ASCII-formatted English-language book, a JPEG image, and a file containing only `0x00` bytes. Shown here are the ciphertext length (Column 2), ciphertext expansion factor (CEF, Column 3), and keystream expansion factor (KEF, Column 4). As expected, CEF ≈ 2 and KEF ≈ 5.2. The right half of this table (Columns 5–7) reports two-point Kolmogorov–Smirnov tests comparing the byte-distribution among the different ciphertexts, and also between each ciphertext and a file of 2560012 uniformly-distributed random bytes ('Random'). The non-significant *p*-values (Column 7) suggest that the ciphertexts do not statistically differ from one another, nor from random bytes

Background

A time-memory tradeoff is a general-purpose probabilistic method for solving certain problems in cryptology and computer science, like inverting a one-way function [16]. Consider $Y = E(X, \text{key})$ which uses one of N possible keys to encrypt X. If no direct way to determine E^{-1} is known, then the cryptanalyst can instead attempt a brute-force solution. In one such approach, the adversary chooses a likely string (e.g., X' = `"Dear Sir or Madam:"`), and encrypts this string under every possible key in advance. The resulting N pairs $\{\text{key}_i,\ E(X', \text{key}_i)\}$ are stored in a large table. Upon intercepting the ciphertext, the actual key can then quickly be found by searching Y for any sub-string that matches an $E(X', \text{key}_i)$ in the table. An alternative approach assumes that the attacker knows some of the plaintext. The corresponding portion of Y is then decrypted under every possible key until the result matches this known plaintext.

Either way, it would seem that brute-forcing the key requires N memory units (to store the table) or N time units (to perform the decryptions), implying that such an attack could be foiled by choosing a large-enough N. However, this need not be the case: TMDTO attacks can efficiently cover enough of the search-space that the probability of success becomes $\gg 0$ while the complexity remains $\ll N$ [29].

TMDTO Attacks

TMDTO attacks against stream ciphers proceed in two phases: a *precomputation* phase, during which one or more tables are constructed from a set of randomly chosen KSG-states; and a *realtime* phase, during which the table(s) are searched for fragments of known keystream. Tradeoff curves involve several parameters [27]:

- $N = 2^n$ is the search-space. For block ciphers, $N = |\mathcal{K}|$, the size of the key-space. For stream ciphers, $N = |\mathcal{S}|$, the number of possible KSG-states;
- P is the time required for the precomputation phase;
- M is the amount of memory required to store the precomputed table(s);
- T is the time required to complete the realtime phase;
- D is the amount of plaintext known to (or chosen by) the attacker;

For PudgyTurtle, one more parameter is also useful:

- D' is the amount of realtime data. For binary-additive stream ciphers, $D' = D$. For PudgyTurtle, however, $D' \geq D$.

Distributed computing can improve the efficiency of many tradeoffs. These effects can be described with another parameter, W, representing the number of parallel processors [27].

For block ciphers, Hellman proposed a tradeoff of $TM^2 = N^2$, using one chosen plaintext ($D = 1$) and a very long precomputation phase ($P = N$) [26]. One reasonable point on this curve is $M = T = N^{2/3}$. Oechslin's 'rainbow table' method [44] somewhat improves Hellman's tradeoff and reduces its need for time-consuming disc-access operations [37].

For stream ciphers, Babbage [5] and Golić [22] independently developed the 'BG-attack', whose tradeoff of $TM = N$ (with $P = M$ and $T \leq D$) arises from the Birthday Paradox. Here, the point $M = T = N^{1/2}$ appears more efficient than Hellman's $N^{2/3}$. However, direct comparisons can be misleading: one attack targets block ciphers, and the other stream ciphers; one uses a single chosen plaintext/ciphertext pair, and the other exploits more realtime data; one uses multiple tables, and the other just one; and so on.

Biryukov and Shamir adapted some of Hellman's methods to create another TMDTO attack against stream ciphers, which accounts for D in detail [12]. The tradeoff of this 'BS-attack' is $TM^2D^2 = N^2$, with $P = N/D$ and $D^2 \leq T \leq N$. For example, one point on this curve, assuming $N \approx 2^{100}$, is $P = T = N^{2/3} = N^{66}$, and $M = D = N^{1/3} = N^{33}$. The BS-attack uses many tables, all related through a simple function like bit permutation.

The goal of both the BG- and BS-attacks is to recover a KSG state. However, TMDTO attacks designed to recover the secret key/IV combination have been proposed by Hong and Sarkar [30] and discussed by Dunkelman and Keller [18]. Tradeoffs in these approaches take the general form of $TM^2D^2 = N^2V^2$, where V is the number of IV's [27].

Another improvement in TMDTO attacks is sampling [13], whose main idea is to limit the attack to a smaller space of *special points* (e.g., KSG states that begin with a certain number of 0's in a row) [4]. Limiting the precomputed table to these points speeds up the realtime phase, since a table-search is only required when the known keystream fragment also happens to start with this string. This method offers different advantages against different ciphers: the tradeoff curve itself may change; its range of parameters may expand; and/or practical speedups (e.g., fewer disk-access operations) may become possible [31, 51].

The BG-Attack

Here we describe in detail the original BG-attack [5, 22]. During the precomputation phase, M unique n-bit starting states S_i are chosen. Each of these is used to initialize the KSG, after which its *prefix*, $e(S_i)$ (i.e., the first n bits of keystream) is computed. The $\{S_i, e(S_i)\}$ pairs are then stored in a $M \times 2$ table, sorted by prefix.

During the realtime phase, it is assumed that the adversary possesses $D + n - 1$ bits of known plaintext, X'. From these data, the known keystream, K', is obtained by XOR'ing X' and the ciphertext at the appropriate position. Then, starting at bit-offset $a = 1$, an n-bit sliding window is applied to K' to produce a known keystream *fragment*, $K'(a) = \{k_a, k_{a+1}, k_{a+2}, \dots, k_{a+n-1}\}$. The table is searched for any prefixes that match this fragment. If none are found, the sliding-window is advanced by one position, and the table is searched for $K'(a + 1)$—a process which may be repeated up to D times. If a matching prefix $e(S')$ is found, then its paired state, S', likely reflects the KSG at some point during encryption. If keystream obtained by seeding the KSG with S' correctly decrypts the relevant portion of Y into X', then the attack succeeds.

The tradeoff curve $TM = N$ suggests that the original search-space can be covered more efficiently than exhaustive search. For example, time and memory resources can be balanced by choosing $T = M = D = \sqrt{N}$. More generally, letting $M = 2^m$ and $T = 2^t$, other tradeoffs can also be made, subject to $m + t \sim n$.

The BS-Attack

Biryukov and Shamir's method (the so-called 'BS-attack') expands Hellman's time-memory tradeoff for block ciphers into the realm of stream ciphers [12]. Unlike Hellman's original idea, however, which assumed a single block of chosen plaintext ($D = 1$), the BS-attack allows attackers to take full advantage of D bits of known plaintext.

Since D may be constrained by factors external to the cryptosystem itself, it is taken as a predetermined 'given' from which the other parameters are calculated. After specifying D, the cryptanalyst next chooses m and t (explained below) which satisfy Hellman's 'matrix-stopping rule': $mt^2 = N$.

During the precomputation phase, t/D tables are constructed, each of dimension $m \times 2$. The first column's entries, called 'start points' (SP_i), are m unique, randomly selected n-bit KSG-states. The second column's entries, called 'end points' (EP_i), are obtained by applying a function t-many times to each corresponding start-point:

$$\mathrm{EP}_i = f \circ f \circ f \ \dots \ \circ f(\mathrm{SP}_i) = f^{(t)}(\mathrm{SP}_i)$$

where $f : \{0,1\}^n \rightarrow \{0,1\}^n$ is explained below. The intermediate results of this composition of functions are called a *Hellman chain*. To save memory, only the first and last links of each chain need to be stored, but—if required—any link can be regenerated from the start-point. Each row thus 'covers' t keys, and an m-row table covers mt keys while only requiring $m \cdot 2n$ bits of storage.

The function $f(S)$ is itself composed of two other functions, e and r, where $e(S)$ is the first n bits of keystream (the 'prefix') produced by the KSG from state S, and r changes this prefix in some simple way, like permuting its bits or XOR'ing it to a constant. Each table has a unique version of r, so

$$f_z(S) = r_z \circ e(S)$$

refers to the version of f used in table z, where $z = 1, 2, \dots, t/D$.

During the realtime phase of the BS-attack, known keystream K' is split into successive n-bit fragments, $K'(a)$, where $a = 1, 2, \dots, D$. For each fragment, the search begins by checking whether or not $r_z(K'(a))$ matches an end-point of any table. If no matches are found, then the adversary modifies the search-target by one application of f, and now searches the end-points for $f_z(r_z(K'(a)))$. The attacker may repeat this, searching through a so-called *realtime Hellman* *chain* by applying f up to t times. If still no match has been discovered, then the next known-keystream fragment, $K'(a+1)$, is processed the same way, until all D fragments have been tried.

When a match is found, the attacker wishes to find the KSG-state, S', for which $K'(a)$ is the prefix: $K'(a) = e(S')$. The first step is to regenerate the appropriate precomputed chain. For example, suppose that the α-th realtime application of f matches the i-th end-point of the z-th precomputed table:

$$f_z^{(\alpha)}(r_z(K'(a)) = \mathrm{EP}_i$$

The adversary then reconstructs the i-th precomputed chain (by $(t-\alpha)$ applications of f_z) to 'meet' the beginning of their realtime chain $f_z^{(t-\alpha)}(\mathrm{SP}_i) = r_z(K'(a))$. This holds because $\mathrm{EP}_i = f^{(t)}(\mathrm{SP}_i) = f^{(\alpha)}f^{(t-\alpha)}(\mathrm{SP}_i) = f^{(\alpha)}(r_z(K'(a))$. Finally, the desired result is the precomputed-chain state immediately preceding this one:

$$S' = f_z^{(t-\alpha-1)}(\mathrm{SP}_i)$$

The cryptanalyst knows that the first n bits of keystream generated by KSG-state S' will equal $K'(a)$, since the attack has been set up so that $f^{(t-\alpha)}(\mathrm{SP}_i) = f(S') = r(e(S')) = r(K'(a))$ and therefore $e(S') = K'(a)$. If this new keystream correctly decrypts the message, the attack succeeds. Otherwise, a false-alarm has been discovered, and the attack continues.

The BS time-memory-data tradeoff can now be appreciated in more detail. Taking each table-search as one 'time-operation', the BS-attack requires searching t/D different tables, for one of D known keystream fragments, and repeating each search for t applications of f, so that $T = (t/D) \cdot D \cdot t = t^2$. For t/D tables containing m rows each, the memory requirement is $M = mt/D$. Thus, from Hellman's matrix-stopping rule $N = mt^2$, the BS-tradeoff is

$$N^2 = (mt^2)^2 = (mt)^2 t^2 = (MD)^2 T = TM^2D^2$$

It is important to note that Biryukov and Shamir's approach does not *require* multiple tables. Rather, the number of tables (t/D) just factors into the tradeoff: using one table means performing the attack with a relatively bigger table and relatively shorter realtime phase, for a given D. The toy cipher used in Sect. 9 has low enough computational and memory requirements that a 'one-table' tradeoff (e.g., $m = t = D \sim N^{1/3}$) can be implemented as reasonably a multi-table tradeoff.

PudgyTurtle and Collision Attacks

This section describes some of the challenges associated with TMDTO attacks against PudgyTurtle. As we have seen, what the adversary has is *known plaintext* X', but what the attacker actually needs is *known keystream*, K'. This observation reveals two (sometimes unstated) assumptions behind TMDTO attacks:

- *Known plaintext equals known keystream.* With binary-additive stream ciphers, K' can be easily obtained by XOR'ing the intercepted ciphertext with X'.
- *Known keystream is contiguous, or at least predictably spaced* [27]. TMDTO attacks involve successively applying a sliding window to K', thereby obtaining targets to search for within the precomputed table(s). It is assumed that an n-bit window produces n bits of useful data.

With PudgyTurtle, neither assumption holds. First, because the plaintext–keystream interaction during encoding is probabilistic, a single known plaintext–ciphertext pair is consistent with many different keystreams. Second, because some keystream nibbles are skipped during encryption, each keystream fragment contains irregularly spaced gaps of data which remain unknown to the attacker. By making it harder to obtain K' from X', the realtime phase of TMDTO attacks against PudgyTurtle becomes more difficult.

Central to this discussion is the idea that a particular (X, Y) pair can be consistent with many different keystreams. To see how this is possible, recall from Table 1 that plaintext `0x48` ("H" in ASCII) produced codewords {`0x10`, `0x04`} and ciphertext `0x4327` under keystream `0x53B14230`.

Yet, this same ciphertext could also have resulted from *different encodings* of plaintext `0x48` under different keystreams. For example, the keystream $\{K_1, K_2, 4, K_4, K_5, 8\}$ exactly matches each plaintext nibble on the first attempt (i.e., no failures), making two `0x00` codewords. Therefore, if the masks $(K_1 \| K_2)$ and $(K_4 \| K_5)$ were chosen to be the same as their corresponding nibbles in the original ciphertext (i.e., keystream {`4,3,4,2,7,8`}), then Y will also be `0x4327`:

$$Y_1 = C_1 \oplus M_1 = \texttt{0x00} \oplus (K_1 \| K_2) = \texttt{0x00} \oplus \texttt{0x43} = \texttt{0x43}$$
$$Y_2 = C_2 \oplus M_2 = \texttt{0x00} \oplus (K_4 \| K_5) = \texttt{0x00} \oplus \texttt{0x27} = \texttt{0x27}$$

Similarly, if a nibble within one mask happened to be one bit off, then the same ciphertext would still result if its corresponding discrepancy code was also one bit off. This might happen, for instance, if the second mask $(K_4 \| K_5)$ was `0x26` instead of `0x27`, and the second discrepancy code was 001_2 instead of 000_2—meaning that K_6 would be $X_2 \oplus 0001_2 = \texttt{0x8} \oplus 0001_2 = 1001_2 = \texttt{0x9}$ instead of `0x8`. Thus, keystream {4,3,4,2,6,9} would also produce the same

Table 3 Different keystreams; same ciphertext

Plaintext	Keystream	Codewords	Ciphertext
Actual keystream			
0x48	5,3,B,1,4,2,3,0	0x10, 0x04	0x4327
Other keystreams producing the same ciphertext			
0x48	4,3,4,2,7,8	0x00, 0x00	0x4327
	4,3,4,2,6,9	0x00, 0x01	
	4,2,5,2,7,8	0x01, 0x00	
	4,2,5,2,6,9	0x01, 0x01	
	…		
	5,3,B,1,4,2,7,8	0x10, 0x00	
	5,3,B,1,4,2,6,9	0x10, 0x01	
	5,3,B,1,4,2,5,A	0x10, 0x02	
	5,3,B,1,4,2,3,C	0x10, 0x04	
	…		
	4,B,B,4,2,F,7,8	0x08, 0x08	
	…		

The original keystream (top) transforms plaintext 0x48 into the ciphertext 0x4327. However, other encodings produced by other keystreams can also have the same effect

ciphertext: Y_2 would remain 0x27, but would be calculated as:

$$Y_2 = C_2 \oplus M_2 = \texttt{0x01} \oplus (K_4 \| K_5) = \texttt{0x01} \oplus \texttt{0x26} = \texttt{0x27}$$

Many other keystreams would also produce this same ciphertext. A few examples are given in Table 3.

Tentative Keystream

TMDTO attacks against PudgyTurtle may have to contend with many possible keystreams, rather than just the single 'known keystream' required to attack a BASC. Here, we describe how the adversary builds a set of *tentative keystreams* from the intercepted ciphertext, the known plaintext, a hypothesized encoding, and something called the 'verified sequence'. These tentative keystreams become the input to our new TMDTO attacks against PudgyTurtle.

The Model. Tentative keystreams are based on different *models* of how X' is encoded. Ignoring overflow events for now, each model, $C^j \subset \mathcal{C}$, is a collection of codewords—one failure-counter and discrepancy-code for each nibble of known plaintext:

$$\begin{aligned} C^j &= \{C_1^j, C_2^j, \dots, C_{N_{X'}}^j\} \\ &= \{(F_1^j \| D_1^j),\ (F_2^j \| D_2^j), \dots\ (F_{N_{X'}}^j \| D_{N_{X'}}^j)\} \end{aligned}$$

where $j = 1, 2, \dots, |\mathcal{C}|$. We emphasize that F_i^j and D_i^j are just guesses, not necessarily the *actual* failure-counter (F_i) and discrepancy code (D_i) that produced Y_i from X_i'.

To specify a particular model, C^*, the components of each codeword, $F_i^* \in \{0, 1, 2, \dots, 31\}$ and $D_i^* \in \{0, 1, 2, 3, 4\}$, can either be chosen randomly or taken from a list—perhaps ordered by probability of occurrence. Since discrepancy-codes are equiprobable, the probability of any model can be ranked according to the product of the probabilities of its failure-counters: $\Pr(C^*) = g(F_1^*, p) \times g(F_2^*, p) \times \cdots \times g(F_{N_{X'}}^*, p)$.

As a specific example assuming that X' is 4 nibbles long, we use the following randomly chosen model

$$\begin{aligned} C^* &= \{C_1^*,\ C_2^*,\ C_3^*,\ C_4^*\} \\ &= \{\texttt{0x0B},\ \texttt{0x10},\ \texttt{0x04},\ \texttt{0x09}\} \\ &= \{00001011,\ 00010000,\ 00000100,\ 00001001\} \\ &= \{00001\|011,\ 00010\|000,\ 00000\|100,\ 00001\|001\} \\ &= \{1\|3,\ 2\|0,\ 0\|4,\ 1\|1\} \end{aligned}$$

Thus, failure-counter F_1^* is 1, discrepancy code D_1^* is 3, $F_2^* = 2, D_2^* = 0, F_3^* = 0$, and so on.

Verified sequence. Given our model C^* from above, the next step is to build its *verified sequence*, V^*. This sequence marks which nibbles of the tentative keystream can be predicted by the model, and which ones remain *unknown* (i.e., keystream nibbles that would have been skipped-over and discarded during encoding because they failed to match a plaintext nibble).

Specifically, $V_i^* = \texttt{0xF} = 1111_2$ if the i-th tentative keystream nibble can be predicted by the model, and $V_i^* = \texttt{0x0} = 0000_2$ otherwise. This means that each nibble of X' adds three 0xF nibbles to V^*: two coinciding with the mask, and one positioned where the keystream nibble would have matched the plaintext nibble. The number of intervening 0-nibbles corresponds to the failure-counter. For example, if some failure-counter in the model were 3, its corresponding representation in V^* would be ...FF000F.... Similarly, a failure-counter of 0 would correspond to … FFF … in V^*, and so on. Thus, the Hamming weight of the verified sequence is $h(V) = 3L_{X'} = 12N_{X'}$ bits.

The verified sequence for our model C^* is shown below, where #'s mark the verified-sequence nibbles associated with each mask, F^* shows the progression of each failure-counter, and *'s mark the verified-sequence nibble associated with each plaintext–keystream match:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Masks	#	#			#	#				#	#		#	#		
F^*			0	1			0	1	2			0			0	1
Matches				*					*			*				*
V^*	F	F	0	F	F	F	0	0	F	F	F	F	F	F	0	F

Filling in. Finally, the tentative keystream K'^* is created by filling in the non-zero elements of V^*. To illustrate this process, assume that the known plaintext X' is `"Hi"` (ASCII `0x4869`) with corresponding ciphertext `0xEE7D22C3`.

Since the first ciphertext byte $Y_1 =$ `0xEE` is made by XOR'ing first codeword $(F_1^* \| D_1^*)$ with the first mask $(K_1'^* \| K_2'^*)$, the attacker knows that $(K_1'^* \| K_2'^*) = Y_1 \oplus (F_1^* \| D_1^*) = 0\text{xEE} \oplus (00001_2 \| 011_2) = 0\text{xEE} \oplus 0\text{x0B} = 0\text{xE5}$. Next, because $F_1^* = 1$, the cryptanalyst deduces that one tentative keystream nibble $(K_3'^*)$ was skipped because it failed to match X_1', and that next tentative keystream nibble $(K_4'^*)$ matched X_1' to within one bit. Its value can therefore be filled in by inverting discrepancy code $D_1^* = 3$:

$$\begin{aligned} K_4'^* &= X_1' \oplus d^{-1}(D_1^*) \\ &= X_1' \oplus 2^{D_1^*-1} \\ &= 0\text{x}4 \oplus 2^{3-1} \\ &= 0 \end{aligned}$$

At this point, the first codeword has been used to fill in the first 4 nibbles of tentative keystream $K'^* =$ `E, 5, ?, 0`⋯, where `?` represents the 'unknown' nibble corresponding to $V_3^* = 0$. The complete tentative keystream (shown in the final row of the diagram below) can be constructed by continuing this pattern.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Masks	#	#			#	#				#	#		#	#		
F^*			0	1			0	1	2			0			0	1
Matches				*					*			*				*
V^*	F	F	0	F	F	F	0	0	F	F	F	F	F	F	0	F
	E	5	...													
	E	5	?	0	...											
	E	5	?	0	6	D	...									
	E	5	?	0	6	D	?	?	8	...						
	E	5	?	0	6	D	?	?	8	2	6	...				
	E	5	?	0	6	D	?	?	8	2	6	E	...			
	E	5	?	0	6	D	?	?	8	2	6	E	C	A	...	
K'^*	E	5	?	0	6	D	?	?	8	2	6	E	C	A	?	8

To summarize, we have described how the adversary builds a tentative keystream from one particular 4-codeword model, in conjunction with a 4-nibble known plaintext and its corresponding 4-byte ciphertext. This particular tentative keystream contains 16 nibbles, of which 12 can be predicted ('verified'), and 4 remain unknown. Different models would produce other tentative keystreams, a collection of which become inputs for the realtime phase of our TMDTO attacks against PudgyTurtle.

TMDTOs and PudgyTurtle

After clarifying some terminology, we describe two new TMDTO attacks against PudgyTurtle (i.e., 'modified' versions of the BG- and BS-attacks), and also discuss how these new attacks differ from their original counterparts.

Terminology

During the realtime phase of our attacks, a *hit* refers to any instance in which an n-bit fragment of realtime data (tentative keystream) matches an entry in the second column of the precomputed table. Every hit falls into one of two categories: *high-quality* and *spurious*. High-quality hits are cryptographically significant events and must therefore be investigated further via a test-decryption. Spurious hits, on the other hand, occur by chance and may therefore simply be ignored.

Each high-quality hit has one of two outcomes: a *valid hit* leads to a correct decryption of some portion of Y into X'; a *false-alarm*, on the other hand, means that the test-decryption was not correct.

If any table contains one or more valid hits, the attack is deemed a *success*; otherwise it's a *failure*. Failures happen either when there are no high-quality hits (i.e., all hits are spurious) or when all high-quality hits are false-alarms.

Hamming-Weight Threshold

During the normal BG- and BS-attacks, *all* hits are assumed to be high-quality; none are ignored. During our new TMDTO attacks, however, spurious hits abound. The reason for this that an 'unknown' nibble from the tentative keystream fragment (marked as '?' in the illustrations in Sect. 6.1) could theoretically match *any* similarly located nibble in the precomputed table.

Fortunately, a simple technique allows the attacker to distinguish high-quality hits from spurious ones. A *Hamming-weight threshold*, $\theta \leq n$, is applied to each n-bit fragment of the verified sequence. If the Hamming-weight of a verified-sequence fragment is $\geq \theta$ (i.e., if it has fewer than $n - \theta$ 'unknown' bits), then any hits discovered while investigating its corresponding n bits of tentative keystream are defined as high-quality. Using θ avoids many unnecessary test-decryptions, thereby speeding up our TMDTO attacks considerably.

θ is chosen to reduce the number of total hits down to a more reasonable number of high-quality hits, subject to constraints imposed by the attacker's processing power. In practice,

$$\frac{2}{3}n \leq \theta \leq \frac{3}{4}n$$

appears to generate an adequate number of high-quality hits while also allowing reasonably fast computation. In the numerical experiments below (Sect. 9), choosing θ in this range produced roughly 200–400 high-quality hits per model, with successful attacks taking ~ 1 day to 1 week on off-the-shelf laptops with Intel® I3/I5-generation processors. We emphasize that the exact value of this parameter is not crucial: θ could always be chosen as $n/2$, for instance, if the attacker is willing to wait somewhat longer for their results.

First TMDTO Attack

Our first attack is inspired by the Babbage–Golić method, but differs substantially by its use of multiple keystream models containing partly-unknown data.

PARAMETER SELECTION

Given state-size $N = 2^n$ and D bits of known plaintext, choose table-size $M = N/D$ (e.g., $M = \sqrt{N}$ is a common choice), and Hamming-weight threshold θ;

PRECOMPUTATION PHASE

Choose M unique n-bit KSG states S_i. Starting from each one, produce an n-bit prefix $e(S_i)$, and store the $\{S_i, e(S_i)\}$ pairs in a table.
Note: Depending upon the search strategy (see Sect. 7.3.2), this table may be sorted by prefixes or left un-ordered;

REALTIME PHASE

1. Choose a model, C^j;
2. Calculate the verified sequence, V^j;
3. Fill in the tentative keystream, K'^j;
4. Test K'^j as follows:

(a) Set a, the offset of an n-bit sliding-window, to $a = 1$;
(b) Apply the sliding window to V^j, producing the n-bit *verified sequence fragment* $V^j(a)$;
(c) If $h(V^j(a)) < \theta$, then assume that any hits will be spurious. Instead of searching the table, set $a \leftarrow a + 1$ and return to 4(b);
(d) Assuming $V^j(a) \geq \theta$, apply the sliding window to the tentative keystream to produce n-bit fragment, $K'^j(a)$;
(e) Bit-by-bit multiply $K'^j(a)$ by $V^j(a)$. We denote this by $K'^j(a) \otimes V^j(a)$ and refer to the result as the *verified tentative keystream fragment*. This ensures that any bits left 'undefined' in software implementations are actually set to 0;
(f) Adjust each prefix in the precomputed table the same way, thus creating M *verified prefixes* $e(S_i) \otimes V^j(a)$;
(g) Search the table for any verified prefixes that match the verified tentative keystream fragment. For simplicity, we imagine a sequential (row-by-row) search;
(h) If a matching verified prefix ("high-quality hit") is found, then use its paired state (S') to perform a test-decryption:

- Loadthe KSG with S';
- Generate enoughkeystream to decrypt Y into asmuch of X' aspossible. For standard stream-cipher operation, decryption is assimple as XOR'ing this newly generated keystream with Y (starting at offset a). For PudgyTurtle, however, somecomputational effort is required to determine the 'firstdecryptable' byte of Y, asdescribed in Sect. 7.3.1.
- If the test decryption matchesX',then a valid hit has been discovered: label the attack a success andSTOP.
- If thetest decryption does not match X',then a false-alarm has occurred: return to Step 4(g) to continue-

searching for the same verified tentative keystream fragment,starting from next row of the table;

(i) If no high-quality hits were discovered, or if they were all false-alarms, then try again with the next tentative keystream fragment: set $a \leftarrow a + 1$ and go to 4(b). This can be repeated up to D' times, where $D' = KEF \times D \approx 5.2D$.

5. If the entire tentative keystream has been searched without finding any valid hits, then repeat the whole process with a new model: set $j \leftarrow j + 1$ and return to Realtime phase, Step 1;

Adjusting the State

The output of this attack consists of a putative KSG-state S' and the bit-offset, a, of its corresponding tentative keystream fragment, $K'(a)$. A test-decryption must then be performed to determine whether keystream produced from S' yields a valid hit or a false-alarm. For the standard attacks against a BASC, test decryptions are simple: generate keystream starting from S', XOR this new keystream to the ciphertext at bit a, and then compare the result to the known-plaintext—also starting from a. With PudgyTurtle, however, test decryptions are more complicated. Since PudgyTurtle operates on nibbles (not bits) and 'skips' some of the keystream, the cryptanalyst cannot simply decrypt starting from bit a, but must instead determine A, the offset of the *'first decryptable' ciphertext byte*.

To simplify notation, we drop the superscript j and just consider the model currently being analyzed. Define $Z(i)$ as the number of keystream nibbles required to encode/encrypt the plaintext $\{X_1, X_2, \dots, X_i\}$ up to its i^{th} nibble, using the model $\{C_1, C_2, \dots\} = \{(F_1 \| D_1), (F_2 \| D_2), \dots\}$:

$$Z(i) = \sum_{u=1}^{i} (3 + F_u) = 3i + \sum_{u=1}^{i} F_u$$

The '3' accounts for the extra `0xF`'s in each verified sequence fragment (e.g., $F_1 = 2$ would correspond to verified sequence fragment `0xFF00F`, and therefore $Z(i)$ would equal $F_1 + 3 = 5$, not just two). Thus, encoding/encrypting the *current* plaintext nibble, X_i, uses the keystream fragment

$$K_{Z(i-1)+1}, K_{Z(i-1)+2}, \dots, K_{Z(i)}$$

and encrypting the *next* plaintext nibble, X_{i+1}, uses keystream starting from nibble $K_{Z(i)+1}$.

Since bit a corresponds to nibble $\lfloor a/4 \rfloor$, the new index, A, is found by calculating the smallest $Z(A)$ such that $Z(A) \geq \lfloor a/4 \rfloor$. Practically, it is convenient to decrypt the ciphertext into 'full' plaintext bytes only. For example, the second plaintext byte $(X_3 \| X_4)$ is produced by decrypting the ciphertext from byte Y_3. Decryption starting at Y_4, however, would produce only half of this plaintext byte, which may cause practical difficulties when comparing files. For this reason, we only allow test-decryptions to start from odd-numbered ciphertext bytes: if A turns out to be even, increase it by one.

After this adjustment process, ciphertext starting from byte Y_A can be decrypted into plaintext starting at *nibble* X_A (i.e., plaintext byte $(A + 1)/2$), using keystream

$$K' = K_{Z(A-1)+1}, \; K_{Z(A-1)+2}, \dots$$

This keystream is produced by loading the KSG with S' and then updating the state $4(Z(A) + 1) - a$ times before generating any keystream.

Comparison to Traditional BG-Attack

Although this new attack broadly resembles the method of Babbage and Golić, there are several important differences. First, our new attack requires multiple 'tentative keystreams' rather than a single known keystream. Second, our attack allows for unknown bits in each table-search, something not necessary for the traditional BG-attack. Third, our attack includes a new parameter (Hamming-weight threshold θ) to reduce spurious hits, which are not a significant problem for the original BG-attack. Fourth, once a hit is found, our method requires further adjusting the KSG-state before each test-decryption, which is also not needed during the standard BG-attack. Fifth, sorting takes longer in our attack. Even though there is only one table, each prefix in its second column must be bitwise multiplied by $V^j(a)$ ('verified') before comparison with the current tentative keystream fragment. This operation changes the values of, and therefore the sorted order of, these prefixes each time. Thus, if sorting is used, the table must be *re-sorted* with each new application of the sliding window. (Alternatively, it can be left unsorted, and searched row-by-row.) Finally, our attack's table-search procedure is more involved. Each prefix in the table is unique, but each *verified* prefix need not be. A binary search returns an index of a matching element but not necessarily a particular index of a repeated matching element. For example, suppose that KSG-state S_w is correct, and that the table contains three different prefixes $e(S_u) \neq e(S_v) \neq e(S_w)$ which become identical once they are verified: $e(S_u) \otimes V^j(a) = e(S_v) \otimes V^j(a) = e(S_w) \otimes V^j(a) = P$. A binary search for P might return $e(S_u)$ or $e(S_v)$ instead of $e(S_w)$, incorrectly leading the cryptanalyst to dismiss the hit as a false-alarm after a failed test-decryption using states S_u or S_v. The cryptanalyst must therefore check whether or not the prefix associated with each high-quality hit is unique and, if not, also perform test-decryptions using KSG-states

$$K'(a) \rightarrow \left\{ \begin{array}{ccccccccc} KV_0(a) & \xrightarrow{r} & H_0[0] & \xrightarrow{f} & H_0[1] & \xrightarrow{f} & \cdots & \xrightarrow{f} & H_0[t] \\ KV_1(a) & \xrightarrow{r} & H_1[0] & \xrightarrow{f} & H_1[1] & \xrightarrow{f} & \cdots & \xrightarrow{f} & H_1[t] \\ \vdots & & & \vdots & & & & \vdots & \\ KV_{63}(a) & \xrightarrow{r} & H_{63}[0] & \xrightarrow{f} & H_{63}[1] & \xrightarrow{f} & \cdots & \xrightarrow{f} & H_{63}[t] \end{array} \right.$$

Fig. 1 Tentative Hellman chains. A keystream fragment with 6 unknown bits, $K'(a)$, is used to create 2^6 variant keystream fragments, $KV_i(a)$, where $i = 0, 1, 2, \ldots, 63$. From these variants, the zero-th link of each chain, $H_i[0]$, is obtained by applying Hellman's r-function. Each chain is then extended, by t applications of Hellman's f-function

associated with any other identical verified prefixes. (Alternatively, the attacker can do a simple sequential search through the whole table for each $K'(a)$ fragment, as mentioned above).

TMDTO Attack #2

Our second TMDTO-attack is inspired by the method of Biryukov and Shamir [12, 26]. However, building Hellman chains from an initially uncertain state produces various difficulties not seen in the classical BS-attack, as discussed below.

Variant Keystream Fragments and Tentative Hellman Chains

One way to modify the BS-attack to deal with 'unknown bits' is to use many realtime Hellman chains instead of just one. These chains are constructed from *variants* of each tentative keystream fragment. If fragment $K'(a)$ has u unknown bits, then it will have 2^u variants, denoted $KV_0(a)$, $KV_1(a)$, $\ldots$, $KV_{2^u-1}(a)$. During the realtime phase of our new attack, each variant initiates its own Hellman chain.

The i-th variant of $K'(a)$ is constructed by replacing each of its unknown bits with one bit from the binary expansion of i. Letting $K'(a) = \{k_1, k_2, \ldots, k_n\}$, $V(a) = \{v_1, v_2, \ldots, v_n\}$, and $i = \{b_1, b_2, \ldots, b_{2^u}\}$, then a simple algorithm to build $KV_i(a)$ is:

1. Set counter $s = 1$;
2. For every bit $j = 1, 2, \ldots, n$ of $K'(a)$:
   - If $v_j = 1$, then k_j is unchanged;
   - If $v_j = 0$, then $\{\, k_j \leftarrow b_s$ and $s \leftarrow s + 1\}$;

As a concrete example assuming a 24-bit KSG state, let $K'(a) =$ `0xA0BCDE` and $V(a) =$ `0xFFF0F3`. Since $V(a)$ has six 0-bits (meaning that $K'(a)$ has 6 unknown bits), we construct 2^6 variants as shown below, with unknown bits in boldface:

$V(a)$	1111	1111	1111	**0000**	1111	**00**11	`0xFFF0F3`
$K'(a)$	1010	0000	1011	1100	1101	1110	`0xA0BCDE`
$KV_0(a)$	1010	0000	1011	**0000**	1101	**00**10	`0xA0B0D2`
$KV_1(a)$	1010	0000	1011	**0000**	1101	**01**10	`0xA0B0D6`
$KV_2(a)$	1010	0000	1011	**0000**	1101	**10**10	`0xA0B0DA`
…			…		…		…
$KV_{62}(a)$	1010	0000	1011	**1111**	1101	**10**10	`0xA0BFDA`
$KV_{63}(a)$	1010	0000	1011	**1111**	1101	**11**10	`0xA0BFDE`

During the realtime phase of our attack, these variant keystream fragments are then used to create the initial links of 2^u *tentative Hellman chains*, as illustrated in Fig. 1. From each variant, a $(t + 1)$-link chain is made by defining the 0-th link of the i-th chain as $H_i[0] = r(KV_i(a))$ and each subsequent link as $H_i[\alpha] = f(H_i[\alpha - 1]) = f^{(\alpha)}(H_i[0])$.

Modified BS Attack

Here we describe our second TMDTO attack (a modified version of the Biryukov–Shamir attack), which uses variant keystream fragments and tentative Hellman chains.

PARAMETER SELECTION

Given $N = 2^n$ (the state-space size) and D (the quantity of known plaintext), choose m and t such that $N = mt^2$, and choose Hamming-weight threshold $\theta \leq n$;

PRECOMPUTATION PHASE

1. Create t/D simply-related r-functions, one for each table;
2. Construct t/D different $m \times 2$ tables. The first column contains start-points, $\mathrm{SP}_1, \mathrm{SP}_2, \ldots, \mathrm{SP}_m$; the second contains end-points $\mathrm{EP}_i = f_v^{(t)}(\mathrm{SP}_i)$, where $f_v = r_v \circ e$, for $i = 1, 2, \ldots, m$ rows, and $v = 1, 2, \ldots, (t/D)$ tables.

Note: We drop the v (table) subscript below for convenience, but emphasize that all steps occur for each table;

REALTIME PHASE

1. As in the modified BG-attack, choose a model C^j, and determine its verified sequence V^j and tentative keystream K'^j.
2. For each bit-offset $a = 1, 2, \ldots, D'$, extract an n-bit fragment of both the verified sequence and the tentative keystream, denoted $V^j(a)$ and $K'^j(a)$ respectively. As before, $D' \approx 5.2D$.
3. Let $u = n - h(V^j(a))$ be the number of unknown bits in the current tentative keystream fragment.
4. If $u > n - \theta$ (i.e., too many unknown bits), then increment a and return to Realtime Step 2;
5. Assuming that $u \leq n - \theta$, create variant tentative keystream fragments $KV_i^j(a)$, where $i = 0, 1, 2, \ldots, 2^u - 1$, as described in Sect. 7.4.1.
6. Apply Hellman's r-function to each of the 2^u variant keystream fragments $H_i^j[0] = r(KV_i^j(a))$, thus forming a set of initial (zero-th) links of each tentative Hellman chain $\{H_0^j[0], H_1^j[0], H_2^j[0], \cdots H_{2^u-1}^j[0]\}$;
7. Search the end-points of the precomputed tables for each of these 2^u initial links:
   - If NO match is found, then update the search-targets to $H_i^j[1] = f(H_i^j[0])$ for $i = 0, 1, \ldots, 2^u - 1$, and search the tables again. If no match is found, continue this process t-many times, with the final 2^u search-targets being $H_i^j[t] = f^{(t)}(H_i^j[0])$;

8. If a match IS found (a 'high-quality hit'), determine the desired KSG-state, S', as follows. For concreteness, assume that the 42-nd endpoint (row) of the table matched the 5-th tentative Hellman chain after α applications of f:

$$H_5^j[\alpha] = f^{(\alpha)}(H_5^j[0]) = \mathrm{EP}_{42}$$

(a) First, regenerate the precomputed Hellman-chain until it matches $H_5^j[0]$ by computing $f^{(t-\alpha)}(\mathrm{SP}_{42})$.
(b) Test the predecessor state, $S' = f^{(t-\alpha-1)}(\mathrm{SP}_{42})$ as follows:

⇒ Update S' as described in Sect. 7.3.1 to produce a new KSG-state and ciphertext byte-offset A for test decryption;
⇒ Perform a test-decryption from Y_A using keystream generated from the updated version of S';
⇒ If this decryption matches the known-plaintext starting at X_A ('valid hit'), then the attack succeeds. If not, return to Realtime Step 7 and continue searching.

Comparison to Traditional BS-Attack

Compared to the original BS attack, our modified attack employs multiple realtime Hellman-chains instead of just one. Otherwise, this modified attack differs from the standard BS-attack in mostly the same ways that the modified-BG attack differs from its original counterpart: (1) it uses models to generate tentative keystreams containing 'unknown' bits, leading to many spurious hits, which in turn must be rejected by the inclusion of a Hamming-weight threshold parameter—none of which apply to the standard BS-attack; and (2) the KSG-state, once discovered, must be adjusted before attempting a test-decryption, unlike the original BS-attack. One similarity with its original counterpart, however, is that sorting the precomputed table(s) helps. In our modified attack, end points (EP_i) do not need to be 'verified' (i.e., bitwise multiplied by the verified sequence fragment), as they do in the modified BG-attack. In essence, tentative Hellman chains fix the problem of unknown bits. Thus, quick-sort and binary-search techniques will speed up this attack just as they would the original BS-attack, and more dramatically than the modified BG-attack.

Quantifying the New TMDTO Attacks

How do our new TMDTO attacks compare to the standard BG- and BS-tradeoffs? Since the precomputation phase of these attacks are similar to their original counterparts, we neglect this phase and instead focus on the realtime duration of each attack.

$\hat{T}$ stands for the number of realtime operations, where a 'time operation' is defined as either one table-search or one test-decryption. Normally, test-decryptions are ignored, since only one is required (or perhaps just a few) [28]. With PudgyTurtle, however, the abundance of 'unknown' bits means that most test-decryptions produce false alarms, and therefore should be counted. This parameter can be expressed as

$$\hat{T} = \frac{N_{\mathrm{searches}} + N_{\mathrm{decrypts}}}{P_{\mathrm{valid}}}$$

where

- N_{searches} is the number of table-searches performed per model;
- N_{decrypts} is the number of test-decryptions per model;

Table 4 Modified BG-attack against two contrived plaintexts

	Known plaintext	Verified tentative keystream	Total tentative keystream	Successful tables (out of 1000)	Valid hits per success	False-alarms per valid hit
	EVERY-3					
1	**4096**	12288	12288	952	3.05	0.02
2	1368	**4104**	**4104**	629	1.50	0.07
	EVERY-3-OR-4					
3	**4096**	12288	15000	974	3.70	59.38
4	1368	**4104**	5032	706	1.70	61.76
5	1112	3336	**4096**	605	1.56	63.93

Scenarios were constructed so that the tentative keystream either contained no unknown bits (EVERY-3, upper section of table) or very few unknown bits (EVERY-3-OR-4, lower section of table). Next, attacks were carried out assuming that one parameter was fixed at $\approx \sqrt{N}$ bits (4096 or 4104, in boldface): either $\sqrt{N}$ bits of known plaintext (Rows 1 and 3); or $\sqrt{N}$ bits of verified tentative keystream (Rows 2 and 4); or $\sqrt{N}$ bits of total tentative keystream (Rows 2 and 5). Note that for EVERY-3, there is no difference between the second and third assumption. Each attack used a single tentative keystream model and 1000 precomputed tables. In these contrived scenarios, success was common (> 600/1000 tables), and successful tables contained > 1 valid hit. Unsurprisingly, the attacker enjoyed more success when granted more realtime data (Rows 1 & 3 vs. Rows 2, 4, & 5). False-alarms occurred in both scenarios, but became noticeably more frequent when the known keystream contained unknown bits (rightmost column, lower vs. upper section of table)

- P_{valid} is the probability that a model yields a valid hit (i.e., successful test-decryption). Specifically,

  $P_{\text{valid}} = N_{\text{valid}}/N_{\text{trials}}$

  where N_{valid} is the observed number of valid hits and N_{trials} equals the number of tables used multiplied by the number of models tested per table.

Normally, $N_{\text{valid}} = 1$, since an attack stops once success is achieved. In these experiments, however, some attacks are allowed to run through a predetermined number trials, possibly producing > 1 valid hit.

The standard BG and BS tradeoffs have time-parameters $T_{\text{BG}} = N/M$ and $T_{\text{BS}} = N^2/(M^2D^2)$. We compare these benchmarks to $\hat{T}_{\text{BG}}$ and $\hat{T}_{\text{BS}}$, which represent the number of realtime operations actually observed during the numerical experiments below.

Implementing the TMDTO Attacks

Here, we launch two new TMDTO attacks against PudgyTurtle and discuss their performance in a variety of situations. For the first several attacks, the KSG will be a 24-bit maximal-period, nonlinear feedback shift register (NLFSR) [17], with the following specifications:

- Initial state S_0 is `0xAAAAAA` = $1010...10_2$.
- State $S_t = (s_0, s_1, \ldots, s_{23})$ evolves according to:
  $o(S_t) = s_0$ is the output bit;
  $b = s_0 \oplus s_1 \oplus s_8 \oplus s_9 \oplus s_{15} \oplus (s_7 \cdot s_{18})$ is the feedback bit;
  $\pi(S_t) = S_{t+1} = (s_1, s_2, \ldots, s_{23}, b)$ is the state-update function;

We emphasize that this is not intended to be a secure KSG, but only a 'toy' cipher for illustrative purposes. Its small key-space of $N = 2^{24} = 16777216$ makes for efficient computations (e.g., using the standard tradeoff parameters like $\sqrt{N} = 4096$ or $N^{1/3} = 256$).

For simulations requiring larger-sized KSG's, we use *linear* feedback shift registers (LFSRs) instead of nonlinear ones. The reason for this is pragmatic: maximal-period NLFSRs are difficult to find, and Dubrova's well-known source only goes up to $n = 25$ [17]. Obviously, there are easier ways to break LFSR-based ciphers than a TMDTO attack, but again these examples are for explanatory purposes only.

Modified BG Attack

Below are results of the first new TMDTO attack against PudgyTurtle.

Experiment 1: Contrived Plaintexts

This experiment is designed to confirm the general feasibility of our approach. Modified BG-attacks are performed against two 'contrived' plaintexts, which have been specifically tailored to bias the results toward success by limiting the number of unknown bits:

- The "EVERY-3" plaintext is constructed by taking every third nibble of the actual keystream. This forces each

codeword to be `0x00` and every nibble of the verified sequence to be `0xF`;
- The "EVERY-3-OR-4" plaintext forces every codeword to be either `0x00` (i.e., an exact match on the first attempt) or `0x08` (i.e., one failure followed by an exact match). This is accomplished by comparing the two keystream nibbles after each mask. If they differ by more than 1 bit, the second one is taken as the plaintext nibble, producing codeword `0x08` and adding `0xFF0F` to the verified sequence. If they are within ≤ 1 bit of each other, then the first one is taken instead, producing codeword `0x00` and adding `0xFFF` to the verified sequence. The net result is a verified sequence with mostly `0xF`'s and some `0x0`'s.

This experiment also examines the question, *"how much realtime data is there?"* The usual TMDTO attack against a BASC grants the attacker D bits of known plaintext, which is assumed to also mean $D = L_{X'}$ bits of known keystream. Since PudgyTurtle is not a BASC, however, its $L_{X'}$ bits of known plaintext becomes $L_{K'}$ bits of tentative keystream, of which only $h(V)$ are known (i.e., 'verified' as corresponding to a 1-bit in V), such that $L_{X'} \leq h(V) \leq L_{K'}$. So, does "$D'$ bits of realtime data" mean that the adversary has $D' = D = L_{X'}$ bits of known plaintext, or $D' = h(V)$ bits of verified-sequence, or $D' = L_{K'}$ bits of tentative keystream? Although the answer is open to interpretation, attacks are performed under each of these assumptions.

Table 4 shows modified BG-attacks against both contrived plaintexts, with different values fixed at $\approx \sqrt{N}$ bits (for technical reasons, this may be 4096 or 4104). The value that is fixed is

- $L_{X'} \approx 4096$, in Rows 1 and 3;
- $h(V) \approx 4096$, in Rows 2 and 4;
- $L_{K'} \approx 4096$, in Rows 2 and 5.

(Note: For EVERY-3, $h(V) = L_{K'}$, so Row 2 works for both assumptions). Each row shows the result of a modified BG-attack using one model and 1000 different precomputed tables. Columns 1–3 show the relative sizes of X', $h(V)$, and K', with the fixed value in boldface. Column 4 shows the number of successes. The probability of success increases with more realtime data, being highest for Rows 1 and 3 (i.e., when $D = 4096$ and $D' = L_{K'} = 12,288$). Column 5, the average number of valid hits per success, illustrates that a single table may contain multiple valid hits. False-alarms (Column 6) occur occasionally even when the verified sequence is all 1's (EVERY-3), but become much more likely when the verified sequence contains even a minimal number of unknown nibbles (EVERY-3-or-4).

Table 5 The Hamming-weight threshold

θ	Total hits	High-quality hits (avg)	Valid hits
10	5324798	8025.7	6
12	5324899	5272.6	6
14	5325373	425.6	5
16	5325323	194.6	5
18	5324958	25.9	0
20	5325073	11.9	0
22	5325381	1.0	0
24	5325025	0.3	0

The modified BG-attack was performed using a range of thresholds (θ) for distinguishing high-quality hits from spurious ones. Each attack used the same precomputed table, the same 250 models, a 24-stage NLFSR as the KSG, and assumed that the attacker knows 4096 bits of 'English' plaintext. For each threshold in Column 1, the corresponding number of total (Column 2), high-quality (Column 3, averaged over 250 models), and valid (Column 4) hits are shown. Lower threshold values (θ = 10–12) do not improve efficiency much—thousands of test-decryptions are still required for each model. Mid-range values (θ = 14–16) improve efficiency by reducing the number of high-quality hits (and test-decryptions) while still achieving success. Higher values ($\theta \geq 18$) reduce success—so few high-quality hits are obtained overall that finding any valid hits among them becomes unlikely. In practice, choosing θ so as to produce several hundred high-quality hits afforded a reasonable balance between an attack's computational cost and its likelihood of success

Table 6 How successful is TMDTO attack #1?

Plaintext source	High-quality hits (avg)	Valid hits
English	271.9	4
Image	271.8	2
Zeros	269.5	16

Shown here are modified BG-attacks against 4096 bit samples of three plaintexts (English, Image, and Zeros) encrypted using PudgyTurtle with a 24-bit NLFSR. Each attack used 1 precomputed table, 1000 tentative keystream models, and Hamming-weight threshold $\theta = 15$. Each model produced ≈ 270 high-quality hits (Column 2), and the success rate ranged from 0.2 to 1.6% (Column 3)

For all subsequent experiments, we assume that the attacker has D bits of known plaintext and $D' \approx 5.2D$ bits of realtime data (tentative keystream)—a conservative assumption most advantageous to the adversary.

Experiment 2: Hamming-Weight Threshold

Since the previous experiment used contrived plaintexts which exactly (or nearly) matched the original keystream, all hits were taken to be high-quality rather than spurious. When the plaintext and model are unrestricted, however, spurious hits become more likely. This experiment shows how different values of Hamming-weight threshold θ reduce

Table 7 Two new TMDTO attacks against PudgyTurtle

n	N_{valid}	N_{trials}	N_{searches}	N_{decrypts}	$\hat{T}_{\text{BG}}$	$\hat{T}_{\text{BG}}/T_{\text{BG}}$
Modified BG-attack						
20	19	5000	3302	321.9	953658	931.3
24	27	20000	10040	251.4	7623260	1861.1
28	4	20000	35276	252.1	177640512	10842.3
32	1	20000	121203	293.8	2429935872	37077.9
					$\hat{T}_{BS}$	$\hat{T}_{BS}/T_{BS}$
Modified BS-attack						
20	11	5000	5786870	352.8	2630555904	160556.4
24	7	5000	17562500	268.5	12544834560	191419.0
28	<1[a]	3500	69665100	272.7	>243828817920	>930133.1

Shown here are modified BG-attacks (upper section) and modified BS-attacks (lower section) against English-language plaintext encrypted with a 'toy' cipher based on a nonlinear or linear feedback shift register, and PudgyTurtle. Column 1 shows the KSG state-size. Columns 2 and 3 show the number of valid hits and the number of trials required to obtain them. Columns 4 and 5 show the number of table-searches and test-decryptions required per model (i.e., per tentative keystream). Column 6 shows the realtime duration of each attack, obtained by dividing the number of time-operations per model ($N_{\text{searches}} + N_{\text{decrypts}}$) by the probability of a successful model ($N_{\text{valid}}/N_{\text{trials}}$). Finally, Column 7 shows the ratio of observed attack times to those predicted by the classical BG-tradeoff ($T_{\text{BG}} = N/D$) and BS-tradeoff ($T_{\text{BS}} = N^2/(M^2D^2) = t^2$). Note that this ratio always exceeds 1. One experiment (final row) did not succeed within the pre-specified number of trials. For this case, N_{valid} is reported as < 1, and the number of time-operations as a lower-bound

[a]No valid hits obtained: probability of success $< 1/3500$

the total number of hits to a reasonable number of 'high-quality' hits.

Modified BG-attacks were performed against encrypted English using the same precomputed table and same 250 randomly-chosen models, but a different θ each time. As shown in Table 5, many values of θ can still produce successful attacks, even with substantially fewer high-quality than total hits (Column 3 vs. Column 2). Making θ too small slows down the attack (i.e., more high-quality hits occur than are needed for success), but making θ too big risks missing a valid hit (e.g., when $\theta \geq 18$, there are too few hits overall for success). Attackers choose θ pragmatically, balancing computational resources against the number of high-quality hits (Sect. 7.2). In our experiments, for example, $\theta = 15$ works well for $n = 24$.

Experiment 3. How Successful is TMDTO Attack #1?

Table 6 shows the modified BG-attack carried out against each of the three plaintexts from earlier (English, Image, and Zeros), with one table, 1000 models, $n = 24$, $D = 4096$, and $\theta = 15$. The success rate, P_{valid} ranged from 0.2 to 1.6%. No successful attack produced more than 1 valid hit, but all had ≈ 270 high-quality hits (i.e., false-alarms).

How does the time required by this new attack compare to the usual BG-tradeoff of $T_{\text{BG}} = \sqrt{N} = 4096$? Assuming that $\approx (1 - \theta/n)$ of the tentative keystream fragments exceed the Hamming-weight threshold, N_{searches} may be estimated as $D'(1 - \theta/n) \approx 5.2D(1 - 15/24) \approx 8192$. N_{decrypts} can be estimated as 270, the average number of high-quality hits. Dividing by the probability of success, we estimate the number of realtime operations $\hat{T} = (N_{\text{searches}} + N_{\text{decrypts}})/P_{\text{valid}}$ to be

$$\frac{8192 + 270}{0.016} \leq \hat{T} \leq \frac{8192 + 270}{0.002}$$

or $528875 \leq \hat{T} \leq 4231000$, which exceeds $T_{BG} = 4096$ by more than 100-fold.

Experiment 4. Scaling the Modified BG-Attack

Experiment 3 suggests that our modified BG-attack requires more time than predicted by the original BG-attack. Is this result simply a fluke for $n = 24$, or does it apply to other state-sizes? To address this issue, we repeated the attack for several different values of n, using LFSRs for $n > 25$ as mentioned earlier. Also in this experiment, N_{searches} and N_{decrypts} were counted rather than estimated.

For KSG sizes $n = 20$, 24, 28, and 32, a modified BG-attack was initiated against PudgyTurtle-encrypted English. Each precomputed table had $2^{n/2}$ rows, and allowed the attacker $2^{n/2}$ bits of known plaintext. Hamming-weight thresholds were θ=13 (for n=20); θ=15 (for n=24); θ=19 (for n=28); and θ=23 (for n=32).

The upper section of Table 7 shows the results. The probability of success ($N_{\text{valid}}/N_{\text{trials}}$) ranged from $4/20000 = 0.02\%$ to $19/5000 = 0.38\%$. The number of table-searches (Column 4) scaled with amount of known plaintext (D), while the number of decryptions ('high-quality hits') remained fairly constant in the range 250–350 (Column 5), based on the choices for θ. Our main finding is that the observed number of realtime operations always exceeded that predicted by the standard BG-tradeoff (i.e., $\hat{T}_{\mathrm{BG}}/T_{\mathrm{BG}} > 1$, Column 7). This suggests that our finding is not a one-off result for $n = 24$, but applies more generally to PudgyTurtle.

Modified BS Attack

Similar to the methods of Experiment 4, we also performed our second ('modified BS') TMDTO attack against PudgyTurtle-encrypted English. The KSG and attack parameters were:

n	KSG	θ	D	t	m
20	NLFSR	12	128	128	64
24	NLFSR	15	256	256	256
28	LFSR	19	512	512	1024

The r-function used for Hellman chains was simply the KSG-state XOR'd to the least-significant n bits of a constant,

$$r(S) = (S \oplus R_0) \otimes (2^n - 1)$$

where $R_0 =$ `0x5075646779547572`—sixty-four bits representing the letters "PudgyTur" in ASCII.

For convenience, each attack was carried out using only one precomputed table at a time (i.e., $t/D = 1$), so that $D = t \approx m \approx 2^{n/3}$. However, an attack could be repeated several times with new tables, as summarized by the N_{trials} parameter. Results are shown the lower section of Table 7. Again, as in Experiment 4, we observed that $\hat{T}_{\mathrm{BS}}/T_{\mathrm{BS}} > 1$ (Column 7).

Note that the $n = 28$ attack did not succeed within the prespecified number of trials. In this case, we reported N_{valid} as < 1 and provided a lower-bound on $\hat{T}_{\mathrm{BS}}$ (i.e., if the attack had continued until getting a valid hit, the success probability would be smaller and $\hat{T}_{\mathrm{BS}}$ would be higher). These findings appear robust to variations in P_{valid}: even if this probability was ~ tenfold higher than observed, ratios in Column 7 would still exceed 1.

Limitations of PudgyTurtle

Despite its improved resistance against TMDTO attacks, PudgyTurtle also has some drawbacks related to short plaintexts, side-channel attacks, and variable time and space requirements.

Plaintext–Ciphertext Mismatch

Length differences between a very short plaintext and its ciphertext could potentially leak one byte of keystream.

Table 8 Plaintext–ciphertext mismatch

	Ciphertext bytes				
	`0xAA`	`0xBB`	`0xCC`	`0xDD`	`0xEE`
	Codewords				
#1	(`0xFF`	c_1)	C_2	C_3	C_4
#2	C_1	(`0xFF`	c_2)	C_3	C_4
#3	C_1	C_2	(`0xFF`	c_3)	C_4
#4	C_1	C_2	C_3	(`0xFF`	c_4)

The 5-byte ciphertext (top) results from encrypting a 4-nibble plaintext whose encoding produced one overflow event. The lower section shows each possible location of the overflow event, where (`0xFF` c_i) means that codeword C_i contains the overflow

Consider a one-nibble (4-bit) plaintext $X = X_1$. If the ciphertext is observed to be 2 bytes instead of just one (i.e., $Y = Y_{1,1} \parallel Y_{1,2}$), then the adversary will know that one overflow event has occurred, and that the keystream has the following structure:

$$K = K_1, K_2, \ldots, K_{35}, K_{36}, \ldots, K_{t(1)}$$

The attacker can thus recover the first keystream byte by computing $(K_1 \| K_2) = Y_{1,1} \oplus$ `0xFF`. If, in addition to knowing the plaintext length, the attacker also knows the *value* of X_1, then more information can be inferred. Specifically, the Hamming distance between X_1 and K_j (for $j > 2$) must be $>$ 1, except for K_{35} and K_{36} (which are unrestricted) and $K_{t(1)}$ (whose Hamming-distance is ≤ 1 from X_1). This reduces the number of possible keystreams in an exhaustive search from $16^{t(1)-2}$ down to $11^{t(1)-5} \cdot 16^2 \cdot 5 \approx 0.008 \cdot 2^{3.46 \times t(1)}$. On a practical note, however, even if $t(1)$ takes its smallest possible value of 37, this still leaves 2^{121} possibilities.

What about slightly longer plaintexts? Consider a 4-nibble (2-byte) plaintext which produces the 5-byte ciphertext `0xAABBCCDDEE` in Table 8. Realizing that an overflow event has occurred, the adversary's goal is to determine the *identity* and *position* of one byte of keystream, $KB_a = (K_a \| K_{a+1})$.

There are four equi-probable ways (lower section of Table 8) for five ciphertext bytes to represent a 4-codeword message containing one overflow event. As shorthand, c_i denotes the final byte of the codeword with the overflow event:

$$C_i = \mathtt{0xFF} \mid\mid ((F_i \bmod 32) \| D_i) = \mathtt{0xFF} \mid\mid c_i$$

In Case #1 (i.e., when $F_1 \geq 32$), the attacker knows that $KB_1 = \mathtt{0xAA} \oplus \mathtt{0xFF}$. In Case #2 (i.e., when $F_2 \geq 32$), the attacker could surmise that $KB_a = \mathtt{0xBB} \oplus \mathtt{0xFF}$ occurs at one of 32 geometrically-distributed locations in the keystream, since C_1 could encode any of 32 possible failure-counters. Similarly, for cases #3 and #4, KB_a could be in any of 64 or 96 different positions, respectively. The probability of guessing KB_a declines as the message gets longer and as a moves farther away from the beginning of the keystream.

For example, assuming that the third codeword, $C_3 = (\texttt{0xFF} \| c_3)$ encodes the overflow, the probability of correctly guessing that KB_a equals $Y_3 \oplus \texttt{0xFF} = \texttt{0xCC} \oplus \texttt{0xFF} = \texttt{0x33}$ is

$$\begin{aligned}\Pr(\mathrm{KB}_a = \texttt{0x33}) &= g(F_1, p) \cdot g(F_2, p)\\ &= p^2 \cdot (1-p)^{F_1} \cdot (1-p)^{F_2}\end{aligned}$$

where, as before, $g(F, p)$ is the geometric distribution with F failures and $p = 5/16$, and $a = 1 + (F_1 + 3) + (F_2 + 3)$. For our short message, this probability ranges from as little as $\Pr(\mathrm{KB}_{69} = \texttt{0x33}) \approx 8 \times 10^{-12}$ (when $F_1 = F_2 = 31$) to as much as $\Pr(\mathrm{KB}_7 = \texttt{0x33}) \approx 0.0976$ (when $F_1 = F_2 = 0$).

Importantly, this seemingly high value (0.0976) actually represents the *conditional* probability $\Pr(\mathrm{KB}_7 = \texttt{0xCC}$, given that [$C_3$ contains the overflow] AND [one overflow occurs in 4 encodings]). The true probability, including the *a priori* chance of both conditions, is actually only

$$\begin{aligned}\Pr(\mathrm{KB}_7 = \texttt{0xCC}) &= (0.0976)\left(\frac{1}{4}\right)\sigma^3\,(1-\sigma)\\ &\approx 0.0000016\end{aligned}$$

where

$$\sigma = \sum_{i=0}^{31} p\,(1-p)^i$$

is the probability of a 'no overflow' encoding.

For the general case of a N_X-nibble plaintext producing a $(N_X + 1)$-byte ciphertext, the probability of guessing KB_a's identity and location is:

$$\begin{aligned}\Pr(KB_a = Y_s \oplus \texttt{0xFF}) &= \left(\frac{1}{N_X}\right)\sigma^{N_X-1}(1-\sigma)\prod_{i=1}^{s-1} g(F_i, p)\\ &= \frac{\sigma^{N_X-1}(1-\sigma)p^{s-1}}{N_X}\prod_{i=1}^{s-1}(1-p)^{F_i}\end{aligned}$$

We performed this calculation for various N_X, with the following results:

N_X	Pr(Guessing 1 keystream byte)	
	Best-case	Typical-case
4	1.55×10^{-6}	1.00×10^{-6}
16	3.88×10^{-7}	1.16×10^{-8}
64	9.69×10^{-8}	8.18×10^{-10}
256	2.42×10^{-8}	1.74×10^{-11}
1024	5.88×10^{-10}	2.04×10^{-13}

The middle column is an unlikely 'best-case' scenario in which *every* failure-counter happens to be 0; the rightmost column represents the 'typical' scenario, obtained by 1000 simulations of randomly-chosen, geometrically-distributed failure-counters. As can be seen, the probability diminishes as messages get longer and when failure-counters are chosen realistically.

To summarize, we have quantified the probability that an attacker could guess a single keystream byte given the knowledge that a N_X-nibble plaintext has been encrypted into a $(N_X + 1)$-byte ciphertext. In practice, though, this particular information (i.e., 8 bits of a long keystream) may be of little use in breaking any stream-ciphers with adequate state-size currently in widespread use.

Side-Channel Attacks

If implemented straightforwardly, PudgyTurtle encryption exhibits data-dependent execution times, which could expose it to a timing-based side-channel attack [34]. By comparing timing differences during the encryption of each plaintext nibble, it might be possible to determine F_i, the failure counter that encodes plaintext nibble X_i. Knowing F_i, the codeword C_i only has 5 possibilities instead of $32 \times 5 = 160$. In this case, the maximum number of models to test during a collision attack (i.e., $|\mathcal{C}| = 5^{N'_X}$) might become small enough to fully enumerate. Even so, practical difficulties are still significant: a 16-byte (32-nibble) plaintext would produce $|\mathcal{C}| > 2^{64}$ possibilities.

Standard countermeasures against timing attacks include constant-time execution, blinding, and chunking. Constant-time execution is difficult to achieve in practice, difficult to maintain (i.e., unpredictable changes may occur with CPU firmware updates), and difficult to implement without a performance penalty. Blinding incorporates a random element into encryption so that the execution-time becomes uncorrelated with the plaintext or key, but also adds complexity to the algorithm. Chunking (also called *bucketing*) breaks a large, variable-length computation into fixed-length pieces which are then returned at predetermined points in the execution cycle [35].

Chunking may be the most appropriate way to harden PudgyTurtle against timing attacks. One idea, for example, would be to always generate the same-sized 'chunk' of keystream for each plaintext nibble, thus making execution time independent of the failure-counter. Each encryption cycle would then proceed as follows, assuming X_i is being encrypted with keystream starting at K_a generated from KSG-state S_{4a}:

1. Generate 34 keystream nibbles K_{a+j} for $j = 0, 1, 2, \ldots, 33$, saving each nibble and its associated KSG-state;
2. Let $(K_a \| K_{a+1})$ be the mask;
3. Calculate $h(X_i, K_{a+j})$ for each $2 \leq j \leq 33$;
4. Calculate $d(X_i, K_{a+j})$ for each $2 \leq j \leq 33$;
5. Pick the first $j \geq 2$ (call it $j = u$) for which the Hamming-distance in Step 3 is ≤ 1;
6. Encode the match between X_i and K_{a+u}. If no match was found (overflow event), then use `0xFF` as the codeword;
7. Encrypt the codeword by XOR'ing it with the mask;
8. Let $a \leftarrow u + 1$; let $i \leftarrow i + 1$; and set the KSG-state to saved state $S_{4(u+1)}$
9. Go to Step 1

Timing attacks are exquisitely dependent on the specific hardware and software used to implement the cryptosystem. Carrying out such an attack is beyond the scope of this paper, and results would in any case be limited to one particular set of implementation choices. We suggest this as an important topic for future research.

The 'chunking' approach described here would keep KEF and CEF the same, but would take longer to encrypt each message. Specifically, let t_g be the time needed to generate one keystream nibble; t_h the time required to calculate the Hamming-distance between two 4-bit numbers; and t_d the time it takes to calculate a discrepancy code. On average, PudgyTurtle needs $PT = 5.2t_g + 3.2t_h + t_d$ time to encrypt each plaintext nibble. With chunking, each encryption cycle would need 34 keystream nibbles, 32 Hamming-distance calculations, and 32 discrepancy code calculations, requiring time $\mathrm{PT}_{\mathrm{chunk}} = 34t_g + 32t_h + 32t_d$.

Keystream generation takes longer than calculating Hamming weights or discrepancy-codes, since the latter two operations could be accomplished by small-sized table lookup. Thus, we assume that $t_h = t_d = x$ and $t_g = \gamma x$, where $\gamma > 1$. From this, it follows that

$$\frac{\mathrm{PT}_{\mathrm{chunk}}}{\mathrm{PT}} = \frac{34\gamma x + 32x + 32x}{5.2\gamma x + 3.2x + x} = \frac{34\gamma + 64}{5.2\gamma + 4.2}$$

Although chunking introduces an execution-time penalty, it actually works better as the gap widens between t_g and t_h (or t_d). For example, PudgyTurtle with chunking would run about 8.4 times slower when $\gamma = 2$; but only 7 times slower when $\gamma = 10$; and just 6.6 times slower when $\gamma = 50$ —approaching the limiting value of $\mathrm{PT}_{\mathrm{chunk}} = 6.54 \times PT$. The exact value for γ, of course, depends on hardware and software implementation details.

Variability

Overflow events make it impossible to know the exact ciphertext length until after encryption. For situations involving fixed-length message fields, PudgyTurtle's variable output-size may be problematic. One solution would be to allow space within a fixed-length string for either overflow events or padding. For example, if L bits of plaintext produce $2L + b$ bits of PudgyTurtle ciphertext on average, users could agree to use a fixed data-block size of, perhaps, $2L + 2b$. In the extremely rare case that this L-bit message required more than $2b$ overflow bits, it would have to be rejected and re-encrypted with a different key; otherwise, any of the $2b$ bits not used to encode overflows would become padding (either a predetermined pattern or random bits).

Not only is the ciphertext length variable, but so is the total time needed for encryption. For many encryption applications (e.g., email, file storage), this may not be problematic. For high-throughput, low-latency applications, however, small fluctuations in the duration of the 'crypto' component could potentially degrade overall system performance.

Conclusions

PudgyTurtle is a way to implement keystream-dependent, variable-length encoding of plaintext before stream-encryption. In some ways, it resembles an encryption mode for stream ciphers, in that its goal is to work along with existing systems. PudgyTurtle is less efficient than normal stream-cipher operation: it produces about twice as much ciphertext and requires about five times as much keystream. However, it is also more robust against TMDTO attacks.

The cryptographic literature contains other approaches aimed at making TMDTO attacks against stream ciphers harder, such as using error-correcting codes [32, 33, 40, 42] and certain encryption modes [23, 24]. PudgyTurtle differs from ECC-based systems in that it is not a randomized-encryption protocol, and does not require an external noise source. It differs from other stream cipher modes by focusing on how keystream is used, rather than on the state-update, re-synchronization, and initialization procedures.

Modified versions of the well-known Babbage–Golić and Biryukov–Shamir time-memory-data tradeoff attacks are proposed and tested against PudgyTurtle, and the extra work required to cope with multiple 'tentative keystreams' and to reject false-alarms is quantified. For toy-cipher KSGs with inner-states of up to 32 bits, our experiments suggest that the number of realtime operations required for TMDTOs against PudgyTurtle exceeds those predicted by the standard BG- and BS-tradeoffs.

Of the two TMDTO attacks against PudgyTurtle, the modified BG-attack did 'better' than the modified BS-attack (i.e., $\hat{T}_{\mathrm{BG}}/T_{\mathrm{BG}}$ exceeds 1 by less than $\hat{T}_{\mathrm{BS}}/T_{\mathrm{BS}}$ does). This is of interest since the BS-attack is considered to be somewhat more complex. While both new TMDTO attacks require more work than their traditional counterparts, the modified BS-attack also includes an 'extra' work-factor (multiple tentative Hellman chains) not present in the modified BG-attack. This scales up the number of table-searches from t^2 to $t^2 \times \overline{2^u}$, where $\overline{2^u}$ is the average number of tentative Hellman chains per model. This translates to factors of 112.6 (n=20; $\theta = 12$), 111.7 ($n = 24$; $\theta = 15$), and 125.5 ($n = 28$; $\theta = 19$), explaining at least some of the relative inefficiency of this attack.

Stream cipher security depends largely upon the details and state-size of the underlying KSG. Since PudgyTurtle works alongside existing ciphers, it is cipher-agnostic: we do not recommend any particular KSG (cipher) over any other. If PudgyTurtle makes TMDTO attacks harder, it then becomes tempting to consider reducing KSG state-sizes. However, we suggest that this is premature. TMDTOs are just one cryptanalytic attack among many, and security against this approach does not imply security against all others. PudgyTurtle itself, or a cipher with which it is used, may still be susceptible to other (non-TMDTO) methods of cryptanalysis. Therefore, we suggest a conservative approach until more research into breaking PudgyTurtle exists: maintain the state-sizes currently specified for existing KSG's, even when using PudgyTurtle.

Acknowledgements The authors acknowledge Albert Mao, MD, PhD for valuable comments regarding side-channel attacks, semantic security, and efficiency issues, and Peter Smith, Emeritus Professor, Keele University, UK for insights regarding output statistics of PudgyTurtle.

Funding No funding was received for this project

Compliance with Ethical Standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Agren M, Hell M, Johansson T, Meier W. Grain-128a: a new version of Grain-128 with optional authentication. Int J Wire Mob Comput. 2011;5(1):48–59.
2. Amin Ghafari V, Hu H. Fruit-80: a secure ultra-lightweight stream cipher for constrained environments. Entropy. 2018;20(3):180.
3. Armknecht F, Mikhalev V. On lightweight stream ciphers with shorter internal states. In: Fast software encryption—22nd international workshop, FSE 2015, Istanbul, Turkey, March 8–11, 2015, Revised selected papers; 2015. pp. 451–470
4. Avoine G, Junod P, Oechslin P. Characterization and improvement of time-memory trade-off based on perfect tables. ACM Trans Inf Syst Secur. 2008;11(4):1–22.
5. Babbage S. Improved "exhaustive search" attacks on stream ciphers. In: European convention on security and detection, 1995, Institution of Engineering and Technology; 1995. p. 161–166
6. Barkan E, Biham E, Keller N. Instant ciphertext-only cryptanalysis of GSM encrypted communication. J Cryptol. 2008;21(3):392–429.
7. Bellare M, Rogaway P. Encode-then-encipher encryption: how to exploit nonces or redundancy in plaintexts for efficient cryptography. In: Okamoto T (ed) Advances in cryptology—ASIACRYPT 2000. Lecture Notes in Computer Science, vol 1976, Springer, London, UK; 2000. p. 317–30
8. Berbain C, Gilbert H. On the security of IV dependent stream ciphers. In: Biryukov A, editor. Fast software encryption. Berlin: Springer; 2007. p. 254–73.
9. Berlekamp E, McEliece R, Van Tilborg HC. On the inherent intractability of certain coding problems. IEEE Trans Inf Theory. 1978;24(3):384–6.
10. Bernstein DJ. Cycle counts for authenticated encryption. In: Workshop record of SASC 2007: the state of the art of stream ciphers; 2007. http://cr.yp.to/papers.html#aescycles. Accessed 25 Apr 2020.
11. Biham E, Dunkelman O. Differential cryptanalysis in stream ciphers. Cryptology ePrint Archive, Report 2007/218, 2007. https://eprint.iacr.org/2007/218
12. Biryukov A, Shamir A. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In: Okamoto T, editor. Advances in cryptology—ASIACRYPT 2000. Berlin: Springer; 2000. p. 1–13.
13. Biryukov A, Shamir A, Wagner D. Real time cryptanalysis of A5/1 on a PC. In: Goos G, Hartmanis J, van Leeuwen J, Schneier B, editors. Fast software encryption. Berlin: Springer; 2001. p. 1–18.
14. Cannière CD, Preneel BT. In: Billet O, Robshaw M, editors. New stream cipher designs, vol. 4986. Lecture notes in computer science. Berlin: Springer; 2008. p. 244–66.
15. Dey S, Roy T, Sarkar S. Some results on fruit. Des Codes Cryptogr. 2019;87:349–64.
16. Dinur I. An algorithmic framework for the generalized birthday problem. Des Codes Cryptogr. 2019;87(8):1897–926.
17. Dubrova E. A list of maximum period NLFSRs. IACR cryptology ePrint archive. Report 2012/166; 2012. https://eprint.iacr.org/2012/166. Accessed 25 Apr 2020.
18. Dunkelman O, Keller N. Treatment of the initial value in time-memory-data tradeoff attacks on stream ciphers. Inf Process Lett. 2008;107(5):133–7.
19. Esgin MF, Kara O. Practical cryptanalysis of full sprout with TMD tradeoff attacks. In: Dunkelman O, Keliher L (eds) Selected areas in cryptography—SAC 2015—22nd international conference, Sackville, NB, Canada, August 12–14, 2015, Revised selected papers, Springer, Lecture notes in computer science, vol. 9566; 2015. p. 67–85

20. Fossorier M, Mihaljević M, Imai H, Cui Y, Matsuura K. An algorithm for solving the LPN problem and its application to security evaluation of the HB protocols for RFID authentication. In: Barua R, Lange T, editors. Progress in cryptology—INDOCRYPT 2006. Lecture notes in computer science, vol. 4329. Berlin: Springer; 2006. p. 48–62.
21. Gendrullis T, Novotný M, Rupp A. A real-world attack breaking A5/1 within hours. In: Oswald E, Rohatgi P, editors. Cryptographic hardware and embedded systems—CHES 2008. Berlin: Springer; 2008. p. 266–82.
22. Golić JD. Cryptanalysis of alleged A5 stream cipher. In: Fumy W, editor. Advances in cryptology—EUROCRYPT '97. Berlin: Springer; 1997. p. 239–55.
23. Hamann M, Krause M. On stream ciphers with provable beyond-the-birthday-bound security against time-memory-data tradeoff attacks. Cryptogr Commun. 2018;10(5):959–1012.
24. Hamann M, Krause M, Meier W. LIZARD—a lightweight stream cipher for power-constrained devices. IACR Trans Sym Cryptol. 2017;1:45–79.
25. Hellman M. An extension of the Shannon theory approach to cryptography. IEEE Trans Inf Theory. 1977;23(3):289–94.
26. Hellman M. A cryptanalytic time-memory trade-off. IEEE Trans Inf Theor. 1980;26(4):401–6.
27. Heys HM. Distributed time-memory tradeoff attacks on ciphers. In: Susilo W, Yang G, editors. Information security and privacy. Cham: Springer; 2018. p. 135–53.
28. Hong J. The cost of false alarms in Hellman and rainbow tradeoffs. Des Codes Cryptogr. 2010;57:293–327.
29. Hong J, Moon S. A comparison of cryptanalytic tradeoff algorithms. J Cryptol. 2013;26(4):559–637.
30. Hong J, Sarkar P. New applications of time memory data tradeoffs. In: Roy B (ed) Advances in cryptology—ASIACRYPT 2005. Lecture notes in computer science, vol 3788, Springer, Berlin, Heidelberg; 2005. p. 353–72.
31. Hong J, Jeong KC, Kwon EY, Lee IS, Ma D. Variants of the distinguished point method for cryptanalytic time memory trade-offs. In: Chen L, Mu Y, Susilo W (eds) Information security practice and experience ISPEC 2008. Lecture notes in computer science, vol. 4991, Springer, Berlin; 2008. p. 131–45.
32. Kara O, Erguler I. A new approach to keystream based cryptosystems. In: The state of the art of stream ciphers: SASC 2008. Workshop record; 2008. p. 205–21.
33. Kara O, Erguler I, Anarim E. In: Proceedings of extended abstracts, international conference on applied and computational mathematics ICACM-2012. Ankara, Turkey: METU; 2012. p. 1–5.
34. Kocher PC. Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems. In: Koblitz N (ed) Advances in cryptology—CRYPTO 96. Lecture notes in computer science, vol. 1109, Springer, London, UK; 1996. p. 104–13.
35. Köpf B, Dürmuth M. A provably secure and efficient countermeasure against timing attacks. In: 22nd IEEE computer security foundations symposium, CSF'09; 2009. p. 324–35
36. Lallemand V, Naya-Plasencia M. Cryptanalysis of full Sprout. In: Gennaro R, Robshaw M (eds) Advances in cryptology–CRYPTO 2015, Part 1. Lecture notes in computer science, vol. 9215. Springer, Berlin; 2015. p. 663–82.
37. Lee GW, Hong J. Comparison of perfect table cryptanalytic tradeoff algorithms. Des Codes Cryptogr. 2012;80:473–523.
38. Mahalanobis A, Shah J. An improved guess-and-determine attack on the A5/1 stream cipher. Comput Inf Sci. 2014;7:115–24.
39. Mantin I, Shamir A. A practical attack on broadcast RC4. In: Matsui M, editor. Fast software encryption. Berlin: Springer; 2002. p. 152–64.
40. McEliece RJ. A public-key cryptosystem based on algebraic coding theory. DSN progress report, Jet Propulsion Laboratory, Pasadena, CA; 1978. p. 114–6.
41. Mihaljevic MJ, Oggier FE, Imai H. Homophonic coding design for communication systems employing the encoding-encryption paradigm. 2010; CoRR. arXiv:1012.5895
42. Mihaljević M, Imai H. An approach for stream ciphers design based on joint computing over random and secret data. Computing. 2009;85:153–68.
43. Mikhalev V, Armknecht F, Muller C. On ciphers that continually access the non-volatile key. IACR Trans Sym Cryptol. 2017;2016:52–79.
44. Oechslin P. Making a faster cryptanalytic time-memory trade-off. In: Boneh D, editor. Advances in cryptology—CRYPTO 2003. Berlin: Springer; 2003. p. 617–30.
45. Peikert C. Lattice cryptography for the internet. In: Mosca M, editor. Post-quantum cryptography: PQCrypto 2014. Lecture notes in computer science, vol. 8772. Cham: Springer; 2014. p. 197–219.
46. Peikert C. A decade of lattice cryptography. Found Trends® Theor Comput Sci. 2016;10(4):283–424. https://doi.org/10.1561/0400000074.
47. Rivest RL, Sherman AT. Randomized encryption techniques. In: Chaum D, Rivest RL, Sherman AT (eds) Advances in cryptology: proceedings of crypto '82. Springer, Boston; 1983. p. 145–63.
48. Saarinen MJO. A time-memory tradeoff attack against LILI-128. In: Daemen J, Rijmen V, editors. Fast software encryption. Berlin: Springer; 2002. p. 231–6.
49. Sarkar P. Modes of operations for encryption and authentication using stream ciphers supporting an initialisation vector. Cryptology ePrint archive. Report 2011/299; 2011. https://eprint.iacr.org/2011/299. Accessed 25 Apr 2020.
50. Shannon C. Communication theory of secrecy systems. Bell Syst Tech J. 1949;28(4):656–715.
51. Standaert FX, Rouvroy G, Quisquater JJ, Legat JD. A time-memory tradeoff using distinguished points: new analysis and FPGA results. Revised papers from the 4th international workshop on cryptographic hardware and embedded systems, CHES '02. Springer, Berlin; 2002. p. 593–609.