SN Computer Science (2023) 4:486
https://doi.org/10.1007/s42979-023-01919-6

ORIGINAL RESEARCH

Characterizing a Time–Memory Tradeoff Against PudgyTurtle

David A. August[1] · Anne C. Smith[2]

Received: 19 February 2022 / Accepted: 14 May 2023

Abstract
PudgyTurtle is not a cipher, but rather an alternative way to utilize the keystream in binary-additive stream-cipher cryptosystems. Instead of modulo-2 adding the keystream to the plaintext, PudgyTurtle uses the keystream to encode 4-bit groups of plaintext, and then to encipher each codeword. One goal of PudgyTurtle is to make time–memory tradeoff attacks more difficult. Here, we investigate one such attack (a modification of the well-known Babbage–Golić method), and show that its time-complexity is harder on average than an analogous tradeoff attack against a standard binary-additive stream cipher; may approach that of a 'brute-force' attack; can be reduced by certain parameter choices; and can be formulated in terms of a probability distribution which is amenable to simulation.

Keywords Symmetric encryption · Stream cipher · Time–memory tradeoff · Coding theory

Introduction

PudgyTurtle uses keystream to encode, as well as to encipher, the plaintext. It is not properly a cipher, but rather a technique that can be incorporated into binary-additive stream cipher systems [2]. Previous work suggested that time–memory attacks against PudgyTurtle were difficult [3, 4], but this conclusion was based on relatively few attacks using a limited range of parameters. Here, we explore one time–memory tradeoff attack in much greater detail, and show how its cryptanalytic cost depends upon various parameters. Our main result is that, while a time–memory tradeoff attack against PudgyTurtle can be more efficient than brute-force, it is typically harder than a 'standard' time-memory attack against a non-PudgyTurtle system with the same inner state size.

First, we provide general overviews of PudgyTurtle and time–memory–data tradeoff attacks ("Notation" and "PudgyTurtle: A Review"). Next, we explain in detail one such attack against PudgyTurtle ("Tradeoff Attacks"), review a measure for its time-complexity, ("Modified Babbage–Golić Attack"), explore how this quantity varies with several different parameters ("Complexity of the Modified BG-Attack"); and show its dependence upon an underlying probability distribution ("Empirical Results"). Finally, we discuss PudgyTurtle in the context of lightweight stream-ciphers ("Lightweight Ciphers").

Notation

Hexadecimal values are prefixed by `0x` and binary values subscripted by 2 (e.g., 254 can be written `0xFE` or 11111110_2). X stands for plaintext, Y for ciphertext, and K for keystream, and all elements of these sequences are 1-indexed. PudgyTurtle operates on 4-bit groups ('nibbles') rather than single bits. Subscripted upper case letters denote groups of bits (e.g., K_i, X_i, and Y_i denote a keystream nibble, a plaintext nibble, and a ciphertext byte). Subscripted lowercase letters denote bits (e.g., the keystream K can be written $K_1, K_2, K_3, \ldots$, where $K_i = (k_{4i-3} \| k_{4i-2} \| k_{4i-1} \| k_{4i}) \in \{0,1\}^4$, $k_i \in \{0,1\}$, and '$\|$' denotes concatenation. The length of binary sequence Z is $|Z|$ (bits) or $N_Z = |Z|/4$ (nibbles). Quantities known to the attacker are decorated with primes or double-primes. For example, $X' \in X$ is the known plaintext, and $Y' \in Y$ is the encrypted known plaintext. This notation of convenience maintains the correspondence between indices (i.e., $X'_1, X'_2, X'_3, \ldots \leftrightarrow Y'_1, Y'_2, Y'_3$, etc.)

✉ David A. August
daugust@mgh.harvard.edu

Anne C. Smith
asmith3142@protonmail.com

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

[2] Boston, MA 02114, USA

Published online: 27 June 2023

without having to include the positional offsets of X' within X and Y' within Y.

PudgyTurtle: A Review

This section provides a short overview of PudgyTurtle's encryption and decryption process, its ciphertext expansion, and its keystream consumption. Readers already familiar with this material may wish to skip to Sect. 3.

PudgyTurtle is a cipher-agnostic method that can be used alongside a binary-additive stream cipher. In essence, a keystream-dependent plaintext encoding step is included before the usual XOR-based enciphering step. PudgyTurtle encryption and decryption are

$$Y = \mathcal{P}(X, K)$$
$$X = \mathcal{P}^{-1}(Y, K). \tag{1}$$

Keystream

The keystream used by PudgyTurtle can be from any finite-state machine 'black-box' keystream generator (KSG). This KSG operates on an $n = \log_2(N)$-bit state, which behaves as follows on the ith iteration:

$$S_{i+1} = \pi(S_i)$$
$$k_i = z(\pi(S_i)),$$

where $S_i \in \{0,1\}^n$ and $S_{i+1} \in \{0,1\}^n$ are the current and next states; $\pi : \{0,1\}^n \rightarrow \{0,1\}^n$ is the state-update function; $z : \{0,1\}^n \rightarrow \{0,1\}$ is the output function; and $k_i \in \{0,1\}$ is the output bit. Since z and π are known to all, the sender and receiver must initialize the KSG to state S_0 with a *secret key* and also possibly an *initial value* (IV, such as a nonce or counter). After initialization, a *warm-up phase* (i.e., updating the KSG a number of times but ignoring the output) may also be used to mix S_0 into the state. In this manuscript, however, the notation KSG[S] refers specifically to the 'immediate' sequence of bits $k_1, k_2, k_3, \ldots$ generated from state S—ignoring warm-up. Thus, PudgyTurtle encryption in (1) can also be written as $Y = \mathcal{P}(X, \text{KSG}[S_0])$.

Encryption

To encrypt plaintext nibble X_i with keystream starting at nibble K_j:

1. Set position marker $t(i-1)$ to $j-1$;
2. Create a *mask* from the first two available keystream nibbles: $M = (K_j \| K_{j+1})$;
3. Generate new keystream nibbles (starting with K_{j+2}) until either
   (a) some keystream nibble, denoted $K_{t(i)}$, matches X_i exactly or differs from it by a single bit. If this happens, proceed to Step 4.
   or...
   (b) 32 keystream nibbles have been generated without a match as described in 3(a). If this *overflow event* happens,
   - Output the byte 0xFF $\oplus M$ to mark the event;
   - Let $j \leftarrow j + 34$;
   - Return to Step 1;
4. Encode the plaintext-to-keystream match:
   - Calculate the *failure counter*, F, which represents the number of times (modulo 32) that X_i failed to match a keystream nibble: $F = t(i) - t(i-1) - 3$. Note that F is limited to the range $\{0, 1, 2, \ldots, 31\}$.
     If there are no overflows (the usual case), then $K_{t(i)} = K_{j+2+F}$. For instance, starting with $i = j = 1$, and assuming that X_1 matches the very first keystream nibble after the mask (i.e., K_3), then the failure counter is zero: $F = t(1) - t(0) - 3 = 3 - 0 - 3 = 0$. If instead X_1 had matched K_4, then F would be 1; and so on. If an overflow occurs (e.g., if X_1 did not match the keystream until K_{40}), then F would be $40 - 34 - 3 = 3$.
   - Define the *discrepancy code*
     $$D = \begin{cases} 0, & \text{if } X_i = K_{t(i)} \\ 1 + \log_2(X_i \oplus K_{t(i)}), & \text{if } X_i \neq K_{t(i)}; \end{cases}$$
     D encodes the similarity between X_i and $K_{t(i)}$ onto the set $\{0, 1, 2, 3, 4\}$ (i.e., 0 for an exact match, and 1–4 for a 1-bit mismatch, with the rightmost/low-order bit defined as position #1).
   - Form an 8-bit *codeword*, $C = (F \| D)$, where F is represented as a 5-bit number and D as a 3-bit number (e.g., if $F = 3$ and $D = 4$, then C would be $00011_2 \| 100_2 = 00011100_2 =$ 0x1C);
5. Encipher this codeword by XOR'ing it with the mask to produce a ciphertext byte $Y = C \oplus M$;
6. Output Y.

Table 1 illustrates how the PudgyTurtle process transforms the short (2-nibble) plaintext 0xAB into ciphertext 0x0516, using keystream 0, 1, 2, 3, 4, 5, 6,

Table 1 PudgyTurtle encryption

PLAINTEXT $X = X_1, X_2 =$ `0xA, 0xB`	
KEYSTREAM $K = K_1, K_2, K_3, \dots, K_{10} =$ `0,1,2,3,4,5,6,7,8,9`	
ENCRYPT $X_1 =$ `0xA`	
Make mask	$M = K_1 \| K_2 =$ `0x01`
Find match	$K_3 =$ `0x2` $= 0010_2$ matches `0xA` $= 1010_2$ to within a bit
Failure counter	No failures before K_3 $F = t(1) - t(0) - 3 = 3 - 0 - 3 = 00000_2$
Discrepancy-code	$D = 1 + \log_2(X_1 \oplus K_3)$ $= 1 + \log_2(\texttt{0xA} \oplus \texttt{0x2})$ $= 1 + \log_2(\texttt{1000}_2) = 4 = \texttt{100}_2$
Encode X_1	Codeword $C = F \| D = \texttt{00000}_2 \| \texttt{100}_2$ $= \texttt{00000100}_2 = \texttt{0x04}$
Encipher	$Y_1 = C \oplus M = \texttt{0x04} \oplus \texttt{0x01} = \texttt{0x05}$
ENCRYPT $X_2 =$ `0xB`	
Make mask	$M = K_4 \| K_5 =$ `0x34`
Find match	$K_6 =$ `0x5` $= 0101_2$ fails to match `0xB` $= \texttt{1011}_2$ to within a bit. $K_7 =$ `0x6` $= 0110_2$ fails to match. $K_8 =$ `0x7` $= 0111_2$ fails to match. $K_9 =$ `0x8` $= 1000_2$ fails to match. $K_{10} =$ `0x9` $= 1001_2$ matches
Failure counter	Four failures before K_{10} $F = t(2) - t(1) - 3 = 10 - 3 - 3 = 4 = 00100_2$
Discrepancy-code	$D = 1 + \log_2(X_2 \oplus K_{10})$ $= 1 + \log_2(\texttt{0010}_2) = 2 = \texttt{010}_2$
Encode X_2	Codeword $C = F \| D = \texttt{00100}_2 \| \texttt{010}_2 = \texttt{0x22}$
Encipher	$Y_2 = C \oplus M = \texttt{0x22} \oplus \texttt{0x34} = \texttt{0x16}$
OUTPUT $Y = Y_1, Y_2 =$ `0x0516`	

Plaintext $X =$ `0xAB` is transformed into ciphertext `0x0516` under keystream $K = \{$`0x0, 0x1, 0x2, 0x3, 0x4,` $0x5, 0x6, 0x7, 0x8, 0x9\}$

`7, 8, 9`. Notice that the ciphertext is twice as long as the original plaintext and that about five keystream nibbles are consumed per plaintext nibble. These 'expansion' effects are detailed in "Expansion and Lengths".

Table 2 illustrates overflow events in detail, assuming that X_1 has already been encrypted (using K_1 through K_4) and X_2 is now being encrypted starting at K_5. Each section of the table shows how a failure counter of $F = 2$ would look assuming a different number of overflow events (0, 1, or 2). In the usual situation (no overflows, top), the plaintext-to-keystream match occurs at K_9; with one overflow, the match would occur at K_{43}; and with two overflows, at K_{77}.

Decryption

Each ciphertext symbol $Y = Y_1, Y_2, \dots$ represents an 8-bit byte – not a 4-bit nibble. To decrypt Y_i with keystream starting at nibble K_j:

1. Create a mask $M = (K_j \| K_{j+1})$;
2. Decipher (*unmask*) Y_i to reveal the underlying codeword $C = Y_i \oplus M$;
3. Decode C as follows:

(a) If $C =$ `0xFF`, then an overflow occurred during the original encoding:

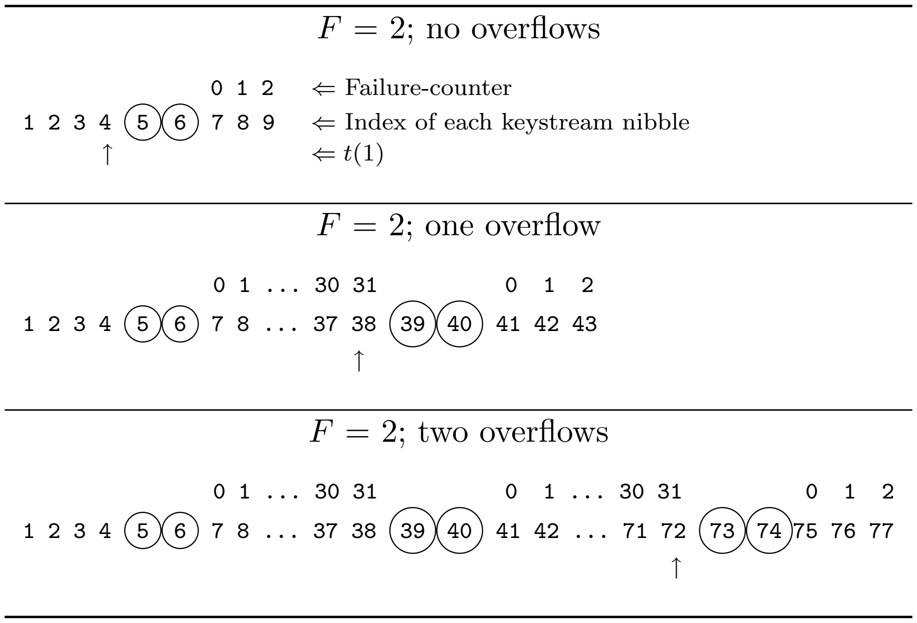
- Generate and discard 32 keystream nibbles;
- Let $j \leftarrow j + 34$;
- Let $i \leftarrow i + 1$;
- Return to Step 1.

(b) If $C \neq$ 0xFF, then

- Extract the failure counter from the codeword's five high-order bits ($F = (C \otimes \text{0xF8}) \gg 3$) and the discrepancy code from its three low-order bits ($D = C \otimes \text{0x07}$), where $\otimes$ is a bit-wise 'AND' and $\gg$ is the right-shift operator.
- Generate $F + 1$ new keystream nibbles. The last of these, $K_{t(i)}$, is the one that matched the plaintext nibble to within a bit;
- Recover the plaintext nibble from $K_{t(i)}$ by inverting D

$$X = \begin{cases} K_{t(i)} & \text{if } D = 0; \\ K_{t(i)} \oplus 2^{D-1} & \text{if } D \neq 0. \end{cases}$$

Table 2 Overflow events



Three examples of how the same failure counter ($F = 2$) could occur with different numbers of overflow events, assuming that the second plaintext nibble is being encrypted under keystream $K_5, K_6, K_7, \ldots$ (i.e., X_1 already consumed K_1 through K_4). Each section above has three rows, showing (respectively) the failure counter; the index of each keystream nibble; and an arrow under the position marker, $t(1)$. Circled indexes are nibbles used in masks. TOP: with no overflows, the plaintext nibble matches K_9, which occurs two failures beyond mask $(K_5 \| K_6)$. Thus, the $F = t(2) - t(1) - 3 = 9 - 4 - 3 = 2$. MIDDLE: with one overflow event, F reaches 31 and re-sets back to zero at K_{41} after a new mask $(K_{39} \| K_{40})$; and $F = 43 - 38 - 3 = 2$. BOTTOM: with two overflows, F re-sets back to zero twice: once at K_{41} and again at K_{75}, which means that $F = 77 - 72 - 3 = 2$. The actual number of failures before the plaintext-to-keystream match could be 2, 34, or 66, all of which equal 2 (modulo 32)

4. Output plaintext nibble X

Expansion and Lengths

The keystream-dependent 'matching' process described in Sect. 2.2 adds an element of unpredictability to PudgyTurtle. Unlike randomized encryption schemes [42], however, this unpredictability depends only upon the secret key: a message encrypted twice with the same key will produce the same ciphertext both times, but the exact ciphertext length can only be guessed on average (before encryption)—not known with certainty.

Specifically, the failure counter behaves like a geometrically distributed random variable, representing one 'success' (i.e., a plaintext-to-keystream match) following zero or more 'failures' (i.e., unsuccessful matching attempts)

$$\Pr\{F = f\} = (1 - p)^f p.$$

In PudgyTurtle, f is limited to the set $\{0, 1, 2, \cdots, 31\}$, so that a 5-bit representation of F can be uniquely decoded. Probability $p = 5/16$ reflects the five possible ways in which two nibbles can 'match'—either they are exactly equal or one of the four 1-bit mismatches.

Whenever a successful match requires > 32 keystream nibbles (an overflow event), an extra byte is inserted into Y as a marker. The probability of such events is

$$\Pr\{F > 31\} = 1 - \sum_{f=0}^{31} (1 - p)^f p \approx 6.0248 \times 10^{-6},$$

suggesting that one overflow occurs ≈ every 80,583 ciphertext bytes (644,664 ciphertext bits).

A standard binary-additive stream cipher system produces ciphertext Y by bit-wise addition of plaintext X to keystream K, so that $|Y| = |K| = |X|$. For PudgyTurtle, these sequences do not all have the same length:

- Overflows mean that the exact ciphertext length is not known until after encryption. However, because overflows are uncommon, $|Y| \approx 2|X|$.
- The keystream-dependent encoding process means that the exact amount of keystream required is not known until after encryption. Since the geometric distribution's mean value is $1/p = (5/16)^{-1}$, approximately 3.2 keystream nibbles will be needed for each successful plaintext-to-keystream match. PudgyTurtle's encryption process also consumes two more keystream nibbles for each mask. Thus, on average, $1/p + 2 = 5.2$ keystream

nibbles are needed to encode and encrypt each plaintext nibble, and $|K| \approx 5.2|X|$.

These values are termed the ciphertext expansion factor (CEF) and keystream expansion factor (KEF). The PudgyTurtle configuration discussed in this manuscript uses 4-bit plaintext symbols (nibbles), 8-bit codewords, 5-bit failure counters, and 3-bit discrepancy codes, which means that CEF ≈ 2 and KEF ≈ 5.2. Note, however, that PudgyTurtle can also be implemented with the other lengths for the codeword, failure counter, etc. in which case CEF and KEF may also differ [4].

Tradeoff Attacks

This section describes the cryptanalytic technique of time–memory–data tradeoff (TMDT) attacks. This method is a powerful, generic tool which does not depend upon system-specific design flaws or weaknesses (e.g., unintended algebraic structure, statistical correlations among keystream bits, inadequate mixing of the secret key into the KSG-state, etc.). TMDT attacks assume that the cryptanalyst has the ciphertext Y, known plaintext X' of size $|X'| \leq |X|$; and the KSG algorithm (functions π and z).

TMDT attacks offer a compromise between two naive cryptanalytic strategies: the 'time extreme' and the 'memory extreme'. The former involves decrypting the ciphertext under every possible secret key and checking whether the result matches the known plaintext. This requires $\mathcal{O}(N)$ time but little memory. The latter involves encrypting the known plaintext under every possible key *in advance*, storing this information in a large table, and then finding the secret key by searching the table for something that matches a sub-string of the intercepted ciphertext. This requires $\mathcal{O}(N)$ memory but little time. TMDT attacks forge a middle-ground: they use more memory than the time-extreme and take longer than the memory-extreme, but their combined time–memory resource requirements can still be $\ll \mathcal{O}(N)$.

The BG-Attack

Although the first cryptanalytic time–memory tradeoff proposed by Hellman targeted block-ciphers [29], these ideas were adapted for use against stream-ciphers as well. Early research by Babbage [5] and Golić [24] led to what has become known as the 'BG-attack'. This attack is conducted in two phases:

1. During the *precomputation* phase, an $M \times 2$ table is constructed,[1] each row of which contains a unique, randomly chosen n-bit KSG-state S paired with its *prefix* P. In this context, 'prefix' means the first n bits of keystream generated when the KSG starts from state S.
2. During *real-time* phase, a 'testing' keystream $X' \oplus Y'$ is constructed, and successive n-bit fragments of this keystream are searched for among prefixes in the table. If a matching prefix, P_h is discovered (a *hit*), then its paired KSG-state, S_h, likely occurred during encryption. This hypothesis can be checked via a *test-decryption* (i.e., using keystream KSG[S_h] to decrypt a segment of Y'). If the test-decryption correctly reproduces the corresponding segment of X', then the attack succeeds and the remaining message can be read; if not, a *false alarm* has occurred, and the attack continues with the next n-bit fragment of testing keystream. Typically, false alarms are rare in the BG-attack.

An important but subtle point is that the word 'keystream' in the description above actually has three different meanings:

- The 'actual keystream' obtained from the initial state, K^A = KSG[S_0], which the sender uses to encrypt the plaintext;
- The 'testing keystream' obtained by XOR'ing the known plaintext to its corresponding ciphertext, $K^T = X' \oplus Y'$, which the attacker uses as a source of n-bit targets to search for within the table;
- The 'regenerated keystream' obtained by initializing the KSG to the state discovered by the hit, K^R = KSG[S_h], which the attacker uses for the test-decryption.

For a standard BG-attack against a binary-additive stream cipher, these three keystreams are equivalent: $K^T = K^R = K^A$, at least starting from some bit-offset. For PudgyTurtle, as we shall see, this is *not* the case.

TMDT attacks against various stream-ciphers like A5/1, GSM, and LILI-128 and others have been proposed [15, 30, 35, 41, 43]. Many researchers have also refined and improved the original BG-attack. For example, Oeschlin suggested multiple reduction-functions within a single 'rainbow table' [40]; Dunkelman and Keller incorporated the IV into a tradeoff framework [21]; Wagner and others have discussed using distinguished points [11]; and Biryukov and Shamir proposed using multiple tables so as to better leverage available data [10].

In previous work, versions of both the BG- and Biryukov–Shamir (BS) attacks were adapted for use against PudgyTurtle. We refer to these as the mBG (modified BG) and mBS (modified BS) attacks. While both modified attacks were less efficient than their original counterparts,

[1] Since this table is constructed offline in advance, the time required for this task is *not* typically factored into the cost of the BG-attack.

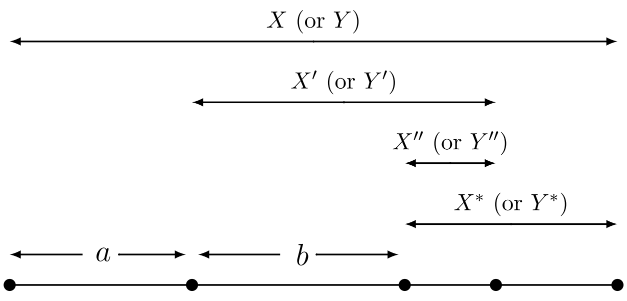


Fig. 1 *Relationships among segments of plaintext (or ciphertext).* The mBG-attack uses known plaintext $X' \in X$ and its corresponding ciphertext $Y' \in Y$ to recover a KSG-state (S'') that correctly decrypts some segment of ciphertext $Y'' \in Y'$ into its corresponding segment of known plaintext $X'' \in X'$. The attacker can then decrypt the rest of the ciphertext (Y^*) even beyond the known plaintext. For notational convenience, all of these segments are all indexed from 1 (i.e., for some indices a and b [bottom], $X'_1 = X_a$, and $X''_1 = X'_b = X_{a+b-1}$, and similarly for analogous segments of Y). mBG: modified Babbage–Golić; KSG: keystream generator

the mBG-attack was faster than the mBS-attack [3]. This was somewhat surprising, since the original BS-attack may be considered an improvement on the original BG-attack. However, several reasons for this observation were proposed (e.g., the need for multiple Hamming-chains, and the relative lack of advantage of binary-search algorithms vs. a simple row-by-row search). Because it was shown to be the better choice, this manuscript focuses exclusively on the mBG-attack.

Modified Babbage–Golić Attack

The modified BG-attack is a tradeoff-based state-recovery attack whose goal is to find a KSG-state S'', such that $X'' = \mathcal{P}^{-1}(Y'', \text{KSG}[S''])$ for some subset of known plaintext $X'' \in X'$ and its corresponding ciphertext $Y'' \in Y'$. The remainder of the original message (X^*) extending beyond X'' can also be read. Relationships among these different segments of text are diagrammed in Fig. 1.

The mBG-attack differs in several important respects from its counterpart, the 'traditional' BG-attack:

- The testing keystream (K^T) in the mBG-attack, from which n-bit search-targets are obtained, is not simply $X' \oplus Y'$, but rather one of potentially many hypothesized keystreams. These 'tentative keystreams' are constructed via a guess-and-determine procedure. Each one is *consistent* with available information (i.e., it encrypts X' into Y'), but may not be *correct* (i.e., it may not decrypt Y' into X', or be obtainable from a KSG-state in the table).
- Each tentative keystream contains *unknown* (undefined) information—keystream nibbles that would have been skipped over and discarded, because they failed to match a plaintext nibble during PudgyTurtle's encoding process.
- False alarms are more common. Since an unknown nibble in K^T could theoretically match *any* nibble in the precomputed table, a fragment of tentative keystream could, therefore, 'hit' many rows of the table by chance, but few (if any) of these coincidental hits will correctly decrypt the ciphertext.

Tentative Keystream

Each tentative keystream represents a different *model* of how X' could have been encoded and enciphered into Y'. The model is just a collection of randomly selected codewords, each one made up of a failure counter and a discrepancy code. Thus, the mBG-attack begins by choosing $N_{X'} = |X'|/4$ failure counters from the set $\{0, 1, 2, \ldots, 31\}$ at random, according to a geometric distribution with $p = 5/16$; and choosing the same number of discrepancy codes uniformly from the set {0,1,2,3,4}. This produces a set of codewords

$$\{C'_1, C'_2, \ldots, C'_{N_{X'}}\} = \{(F'_1 \| D'_1), (F'_2, \| D'_2), \ldots, (F'_{N_{X'}} \| D'_{N_{X'}}).\}$$

Next comes *filling in*, a guess-and-determine process whereby each codeword is used to define three nibbles of tentative keystream: two that make up the mask and one that matches a nibble of known plaintext.

Consider a situation where the attacker knows the correspondence (X'_i, Y'_i) and has a model suggesting that codeword $C'_i = (F'_i \| D'_i)$ produced this correspondence, under tentative keystream starting at K^T_j. First, since Y'_i results from XOR'ing the ith mask and codeword, the attacker concludes that mask $(K^T_j \parallel K^T_{j+1})$ must equal $Y'_i \oplus C'_i$. This fixes the position and identity of two nibbles of tentative keystream. Next, the model's failure counter implies that X'_i would match the tentative keystream on the $(F'_i + 1)$th nibble after the mask. This fixes the location of a third tentative-keystream nibble, $K^T_{t(i)}$, where

$$t(i) = F'_i + j + 2 = F'_i + t(i-1) + 3$$

since $j = t(i-1) + 1$. Finally, the model's discrepancy code D'_i can be used to fix the identity of this third tentative-keystream nibble

$$K^T_{t(i)} = \begin{cases} X'_i & \text{if } D'_i = 0 \\ X'_i \oplus 2^{D'_i - 1} & \text{if } D'_i \neq 0. \end{cases}$$

The end result of filling-in is a segment of tentative keystream

$$\ldots, K^T_j, K^T_{j+1}, ?, ?, \ldots, ?, K^T_{t(i)}, \ldots$$

Table 3 Making a tentative keystream

INPUT	
Known plaintext	$X' =$ `0xAB`
Ciphertext	$Y' =$ `0x0516`
Model	$C'_1, C'_2 =$ `0x12, 0x0C`
FIRST CODEWORD	
Re-create mask	$M = Y'_1 \oplus C'_1 =$ `0x05` $\oplus$ `0x12` $=$ `0x17`
Mask defines two nibbles	$K^T_1 =$ `0x1`, $K^T_2 =$ `0x7`
Find position of keystream nibble that matches X'_1	$t(1) = F'_1 + t(0) + 3 = (C'_1 \otimes \text{0xF8}) \gg 3) + t(0) + 3 = 00010_2 + 0 + 3 = 5$
Define this nibble	$K^T_5 = X'_1 \oplus 2^{D'_1 - 1} =$ `0xA` $\oplus\, 2 =$ `0x8`
Resulting segment of K^T	`1,7,?,?,8`
SECOND CODEWORD	
Re-create mask	$M = Y'_2 \oplus C'_2 =$ `0x16` $\oplus$ `0x0C` $=$ `0x1A`
Mask defines two nibbles	$K^T_6 =$ `0x1`, $K^T_7 =$ `0xA`
Find position of keystream nibble that matches X'_2	$t(2) = F'_2 + t(1) + 3 = 1 + 5 + 3 = 9$
Define this nibble	$K^T_9 = X'_2 \oplus 2^{D'_2 - 1} =$ `0xB` $\oplus 2^{4-1} =$ `0x3`
Resulting segment of K^T	`1,A,?,3`
OUTPUT	
$K^T = 1\ 7\ ?\ ?\ 8\ 1\ \text{A}\ ?\ 3$	

This table shows the guess-and-determine process for ’filling in’ a tentative keystream, starting from a model (codewords $C'_1 =$ `0x12` and $C'_2 =$ `0x0C`), the known plaintext $X' =$ `0xAB`, and its corresponding ciphertext $Y' =$ `0x0516`. Codeword C'_1 is first used to determine the mask `0x17`, which defines two nibbles of K^T. Next, C'_1 is used to define the position and identity of a third nibble of K^T that matches X'_1 to within a bit (namely, K^T_5=`0x8`). This involves splitting C'_1 into its failure counter and discrepancy code (i.e., $C'_1 = \text{0x12} = 00010010_2 = 00010_2 \| 010_2 = F'_1 \| D'_1$), so that $F'_1 = 2$ and $D'_1 = 2$. Notice that the final tentative keystream contains three ’unknown’ nibbles (the third, fourth, and eighth) in addition to six ’known’ nibbles. Nothing can be concluded about these unknowns except that they failed to match their corresponding known-plaintext nibble (e.g., the third and fourth nibbles of K^T could take any value that differed from $X'_1 =$ `0xA` by $\geq$ two bits—namely, `0x0, 0x1, 0x3, 0x4, 0x5, 0x6, 0x7, 0x9, 0xC, 0xD`, or `0xF`)

with three ’known’ nibbles and F'_i intervening ’unknown’ nibbles (represented by ? symbols). All that can be stated about the unknown nibbles is that none of them would have matched X'_i to within a bit.

By doing the same with every codeword, the rest of K^T can be specified. Table 3 illustrates how a tentative keystream would be constructed given $X' =$ `0xAB` and $Y' =$ `0x0516`, using a model whose codewords are `0x12` and `0x0C`.

Verified Sequence

We also define a *verified sequence*, V, as a sequence of 0/1 markers which distinguish known bits of K^T from unknown bits

$$V_j = \begin{cases} \text{0xF} = 1111_2, & \text{if } K^T_j \text{known;} \\ \text{0x0} = 0000_2, & \text{if } K^T_j \text{is unknown;} \end{cases}$$

V contains runs of twelve 1-bits in a row interspersed with variable-length gaps of 0-bits, as shown below

$$\ldots \overbrace{0000\ldots0}^{4F_{i-1}\text{ zeros}} \underbrace{1111\ldots1}_{12\text{ ones}} \overbrace{0000\ldots0}^{4F_i\text{ zeros}}\ \underbrace{1111}_{\text{Matches } X'_i}\ \underbrace{11111111}_{\text{Masks } X'_{i+1}} \overbrace{0000\ldots0}^{4F_{i+1}\text{ zeros}} \underbrace{1111\ldots1}_{12\text{ ones}} \ldots$$

The first four 1’s of each 12-bit run correspond to the bits of K^T that match a known plaintext nibble (e.g., X'_i), and the other eight 1’s correspond bits of K^T that would be used to mask the next (e.g., X'_{i+1}) known-plaintext nibble.

By serving as a bit-mask, V can define unknown bits as zeroes. For example, the model in Table 3 has codewords whose failure counters are 2 and 1, represented by verified sequence `0xFF00FFF0F`. Bitwise-multiplying this with the tentative keystream transforms the partially undefined sequence `1,7,?,?,8, 1,A,?,3` into the fully defined number `0x170081A03`.

Word-Based Notation

The tentative keystream can also be viewed as a sequence of *variable-length ’words’*

Word	.... 1				.... 2			.... 3				
	1 failure				0 failures			2 failures				
Nibble	1	2	3	4	5	6	7	8	9	10	11	12
K^T	K_1^T	K_2^T	?	K_4^T	K_5^T	K_6^T	K_7^T	K_8^T	K_9^T	?	?	K_{12}^T
V	`0xF`	`0xF`	0	`0xF`	`0xF`	`0xF`	`0xF`	`0xF`	`0xF`	0	0	`0xF`

$\downarrow \qquad \searrow$

Word	.... 1													
Nibble	.... 1				.... 2				.... 3				.... 4	
Bit	1	2	3	4	5	6	7	8	9	10	11	12	13	14. . .
K^T	k_1^T	k_2^T	k_3^T	k_4^T	k_5^T	k_6^T	k_7^T	k_8^T	?	?	?	?	k_{13}^T	k_{14}^T . . .
V	1	1	1	1	1	1	1	1	0	0	0	0	1	1. . .

Fig. 2 *'Words' of tentative keystream and verified sequence.* UPPER PANEL: Three words of a tentative keystream generated by a model with failure counters 1, 0, and 2. Row 1 gives the index of, and number of failures in, each word; Rows 2 and 3 provide the index and identity of each tentative keystream nibble (where '?' stands for unknown); and Row 4 gives the accompanying nibbles of verified sequence. LOWER PANEL: Close-up of the beginning of word #1. The first three rows depict the indices of the word, its nibbles, and its bits; the next two rows show individual bits of K^T and V, with unknown bits in K^T (symbolized as '?') corresponding to 0-bits in V and known bits of K^T corresponding to 1-bits in V

$$K^T = W_1, W_2, \dots, W_{N_{X'}},$$

where word W_i contains all $F'_i + 3$ nibbles involved in encoding and enciphering X'_i:

- The first two (known) nibbles are the ith mask;
- The next F'_i (unknown) nibbles are those that failed to match X'_i—including case of an immediate match, where $F'_i = 0$;
- The last (known) nibble is the one that matches X'_i to within a bit.

Failure counters are ≤ 31, so that word lengths are geometrically distributed between 3 and 34 nibbles (12–136 bits) with an average of 5.2 nibbles (20.8 bits).

The verified sequence can also be viewed as a sequence of words, each taking the form `0xFF0...0F`. While their length differs (as above), all words of V share the same Hamming-weight of 12, where Hamming-weight $h()$ is defined as

$$h(U) := \sum_{s=1}^{|U|} u_s$$

for vector $U = (u_1, u_2, \dots, u_{|U|})$ with $u_s \in \{0, 1\}$.

Figure 2 illustrates the structure of the first three words of a hypothetical tentative keystream and its verified sequence, assuming a model with failure counters $F'_1 = 1$, $F'_2 = 0$, and $F'_3 = 2$.

Modified BG-Attack: Detailed Description

Next, the mechanics of the mBG-attack are described in detail, and several definitions are provided. During the precomputation phase, a table with M (state, prefix) pairs is constructed—just like the traditional BG-attack. During the real-time phase, the mBG-attack makes use of successive n-bit fragments of tentative keystream.

DEFINITION. For binary sequence $Z = \{z_1, z_2, \dots\}$, the *fragment* $Z(b) = \{z_b, z_{b+1}, z_{b+2}, \dots, z_{b+n-1}\}$ is an n-bit subsequence starting at bit b.

For example, if keystream $K = \mathrm{KSG}[S]$, then fragment $K(1)$ is the prefix of state S. Each successive n-bit fragment $K^T(b)$ with $b \in \{1, 2, \dots, \ell\}$ and $\ell = |K^T| - n + 1 \approx (5.2)D$ is searched for among the prefixes in the table. However, before searching, all unknown bits must be defined (set to zero) by bit-wise multiplying $K^T(b)$ by $V(b)$ and also multiplying each prefix in the table by $V(b)$.

DEFINITION. For three binary vectors $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ of the same length, we say $\mathbf{A}$ *mimics* $\mathbf{B}$ *under* $\mathbf{C}$ if $\mathbf{A} \otimes \mathbf{C} = \mathbf{B} \otimes \mathbf{C}$.

DEFINITION. A *hit* between tentative-keystream fragment $K^T(b)$ and prefix P_h in the precomputed table means that $K^T(b)$ mimics P_h under $V(b)$.

Importantly, a hit in the mBG-attack does not necessarily imply that $K^T(b)$ equals P_h—only that the *known* bits of the fragment match those bits in the same positions within the prefix. (Unknown bits match automatically, having all been set to zero.)

The more unknown bits, the more likely a hit will occur purely by coincidence. One simple strategy to reduce these coincidences is to reject any hits for which the number of known bits falls below some *threshold*, θ. The attacker's choice of θ has significant implications: at the lowest values ($\theta \approx 0$), the mBG-attack somewhat resembles an exhaustive search/brute-force attack; at the highest values ($\theta \approx n$), the mBG-attack somewhat resembles a 'traditional' BG-attack. ("Extremes of $\boldsymbol{\theta}$" covers this topic in detail.)

Since the number of known bits in $K^T(b)$ is just the Hamming-weight of its corresponding verified-sequence fragment $h(V(b))$, two kinds of hits can be distinguished:

DEFINITION. A *spurious hit* involving $K^T(b)$ means a hit for which $h(V(b)) < \theta$.

DEFINITION. A *high-quality hit* involving $K^T(b)$ means a hit for which $h(V(b)) \geq \theta$.

In the modified BG-attack, many hits are spurious. But even a high-quality hit may not crack the cipher: it still contains $n - \theta$ unknown bits, any of which may mean that the prefix is paired with the 'wrong' KSG-state (i.e., $K^T(b)$ mimics P_h under $V(b)$, but keystream KSG$[S_h]$ generated from P_h's paired KSG-state does not match up with $K^A = \text{KSG}[S_0]$, the actual keystream used to encrypt the message).

Therefore, a *test-decryption* must be performed on each high-quality hit, to determine whether KSG$[S_h]$ will indeed decrypt some portion of Y' into the corresponding portion of X'. This test-decryption need not involve all of Y'—only enough bits to ensure a statistically meaningful comparison with the known plaintext. We refer to this number of bits as Q. Here, Q was chosen as 96 bits. Note, however, that—due to PudgyTurtle's keystream expansion property—decrypting 96 bits requires $\approx (5.2)(96) = 500$ bits of tentative keystream, on average.

Adjustment

During the traditional BG-attack, the test-decryption proceeds directly and immediately from the 'hit'. In other words, if the bth keystream fragment $X'(b) \oplus Y(b)$ produces a hit, then keystream generated by its paired KSG-state in the table will decrypt the ciphertext starting at bit $y_{\omega+b}$ into plaintext starting at bit $x_{\omega+b}$, where ω is the bit-offset of X' within X.

Applying this same strategy to the modified BG-attack, however, becomes problematic: (1) due to keystream expansion, the index of the keystream nibble K_j quickly becomes 'out of synch' with that of plaintext nibble X_i; (2) since PudgyTurtle operates on nibbles rather than bits, decryption requires that b is a multiple of 4; (3) since it is easier in practice to compare strings aligned byte-by-byte, test-decryptions are more efficient when b is a multiple of 8.

Thus, before each test-decryption, the mBG-attack includes an *adjustment* procedure. Given a high-quality hit between tentative keystream fragment $K^T(b)$ and prefix P_h (with associated KSG-state S_h), the goal of this adjustment is to find a new index (ω), such that N_Q ciphertext bytes (starting at byte Y'_ω) can be test-decrypted into N_Q known-plaintext nibbles (starting at nibble X'_ω), where $N_Q = Q/4 = 24$.

Table 4 Adjusting a high-quality hit

Index of 'word' in K^T	Failure-counter	Bit-range of word	Plaintext NIBBLE	Plaintext BYTE	Ciphertext BYTE
1	1	1–16	1	1	1
2	0	17–28	2		2
3	2	29–48	3	2	3
4	1	49–64	4		4
5	1	65–80	5	3	5
6	0	81–92	6		6

UPPER: For each of the first six 'words' ($W_1 - W_6$) of an arbitrary tentative keystream K^T, the word index (Column 1), its associated failure counter (Column 2), and its bit-span (Column 3) are shown. For example, the first failure counter (i.e., 1) represents the pattern $K_1^T, K_2^T, ?, K_4^T$ (where ? is an unknown nibble), which spans $4+4+4+4 = 16$ bits. The next three columns show indices of the known-plaintext nibble (Column 4) and byte (Column 5), and the ciphertext byte (Column 6) corresponding to each word of K^T. LOWER: Adjustment procedure, assuming $K^T(50)$ makes a high-quality hit. Bit #50 lies within word W_4. However, since W_4 is not exactly aligned with a plaintext byte (i.e., it is the second half of byte #2), the word index is incremented by one. Word W_5 starts at bit-offset 65, which means that the original KSG-state S_h should be updated 65–50 = 15 times before the test-decryption. With this adjustment, decrypted ciphertext (starting at Y_5) can be compared to known plaintext (starting at byte #3, made from nibbles X'_5 and X'_6)

A high-quality hit involving fragment$K^T(50)$
Bit-offset $b = 50$ falls within tentative-keystream word W_4 (bits 49–64)
→ W_4 is *not* aligned with the the beginning of a plaintext byte,
→but word W_5 is
→The new bit-offset corresponding to W_5 is $b'' = 65$, and
→ the new KSG-state (S'') is 65–50 = 15 state-updates beyond the original state
→Now, a test-decryption from Y_5 can be compared to X', starting from plaintext byte #3 (i.e., nibbles #5 and #6)

This new index ω defines the first *byte* of an N_Q-byte subset of ciphertext

$$Y'' := \{Y''_1, Y''_2, \ldots, Y''_{N_Q}\} = \{Y'_\omega, Y'_{\omega+1}, Y'_{\omega+2}, \ldots, Y'_{\omega+N_Q-1}\}$$

and the first *nibble* of an N_Q-nibble subset of known-plaintext

$$X'' := \{X''_1, X''_2, \ldots, X''_{N_Q}\} = \{X'_\omega, X'_{\omega+1}, X'_{\omega+2}, \ldots, X'_{\omega+N_Q-1}\}.$$

To decrypt Y'' into X'', the original KSG-state (S_h) must be advanced to a new state, S'', which will produce tentative keystream starting from a new bit-offset, b'', instead of original bit-offset b.

Recall that word W_i of the tentative keystream contains all tentative-keystream nibbles involved in decoding and decrypting ciphertext byte Y'_i and/or encoding and encrypting plaintext nibble X'_i. Since the index of the *final bit* of W_i is

$$\beta(W_i) = \sum_{j=1}^{i} |W_j|$$

the *word* containing bit b will be W_s, where

$$s = \min_{1,2,3,\ldots,N_{X'}} i \;\; \beta(W_i) \geq b.$$

To facilitate string comparisons, each test-decryption should start at the beginning of a plaintext byte. Since plaintext bytes #1, #2, #3,... correspond to plaintext nibbles #1, #3, #5,..., which correspond to tentative keystream words #1, #3, #5,..., the goal can be easily accomplished by adding one to s if it is even. Thus, the index of the *new* word of K^T is

$$\omega = \begin{cases} s, & \text{if } s \text{ is odd} \\ s+1, & \text{if } s \text{ is even.} \end{cases} \tag{2}$$

Using this word of tentative keystream, a test-decryption will begin at the next-nearest whole byte of plaintext. The bit-offset of this new word within K^T will be

$$b'' = \beta(W_{\omega-1}) + 1,$$

and the new KSG-state corresponding to this bit-offset is

$$S'' = \pi^{(b''-b)}(S_h). \tag{3}$$

Thus, KSG$[S''] = K_\omega, K_{\omega+1}, K_{\omega+2}, \cdots$ can be used to test-decrypt $Y'_\omega, Y'_{\omega+1}, Y'_{\omega+2}, \cdots$, and the result compared to X'_ω, $X'_{\omega+1}, X'_{\omega+2}, \cdots$, where conveniently $(X'_\omega \| X'_{\omega+1})$ is aligned with a whole byte of data (namely, the $[(\omega+1)/2]$-th byte).

A worked example is given in Table 4, which adjusts a high-quality hit involving $K^T(50)$ (i.e., bit-offset b=50). The new index is $\omega = 5$, from which the new bit-offset $b'' = \beta(\omega - 1) + 1 = 65$ and new KSG-state $S'' = \pi^{65-50}(S_h) = \pi^{15}(S_h)$ can be found. After this adjustment, a test-decryption using the new keystream KSG$[S'']$ can be applied to the ciphertext starting at Y_5, and then compared byte-by-byte with X' (i.e., known-plaintext bytes #3, #4, #5, etc. corresponding to known-plaintext nibbles (#5, #6), (#7, #8), (#9, #10) etc.).

DEFINITION. For a high-quality hit between fragment $K^T(b)$ and row (P_h, S_h) in the table, $K^R = \text{KSG}[S'']$ is the *regenerated keystream*, where S'' is from Eq. (3).

DEFINITION. $T = \mathcal{P}^{-1}(Y'', K^R)$ is the test-decryption of Y'' under keystream K^R.

DEFINITION. A *correct test-decryption* means that $T_j = X'_{\omega+j} = X''_j$ for every $j \in \{0, 1, 2, \ldots, N_Q\}$ and where ω is from Eq. (2).

DEFINITION. A *valid hit* is a high-quality hit which yields a correct test-decryption.

DEFINITION. A *false alarm* is a high-quality hit which yields an incorrect test-decryption.

If a false alarm occurs, the mBG-attack continues by searching the rest of the table for any other prefixes that mimic $K^T(b)$ under $V(b)$. Once the entire table has been searched, b is incremented by 1, and a table-search is performed on this new target (i.e., $K^T(b+1) \otimes V(b+1)$) until all of the tentative keystream has been utilized. If still no valid hits have been discovered, a new model is chosen, a new tentative keystream is built, and the entire process is repeated.

Complexity of the Modified BG-Attack

This section describes the time-, memory-, and data-complexity of the modified BG-attack. Both the traditional BG-attack and the mBG-attack have the same memory complexity

$$M_{BG} = M_{mBG} = M$$

and the same data-complexity

$$D_{BG} = D_{mBG} = D,$$

where M is the number of rows in the precomputed table, and $D = |X'|$ is the number of bits of known plaintext. (Note: earlier, 'D' was used to symbolize the discrepancy-code, and M the mask, but context should make it clear what is being discussed.) The time-complexity of the two attacks, however, differs

$$T_{BG} \neq T_{mBG}.$$

Traditional BG-Attack: Time-Complexity

For the traditional BG-attack, D bits of known-plaintext data means D keystream fragments, and therefore D table-searches. Thus

$$T_{BG} = D. \tag{4}$$

Most precisely, $T_{BG} = D - n + 1$, but (4) is a reasonable approximation, since usually $D \gg n$. Although this equation is useful in practice, it contains several assumptions:

- Assumption #1: Recall from Sect. 3.1 that the n-bit fragments used as search-targets are actually from the 'testing keystream' $(X' \oplus Y')$, not from the known plaintext itself. Equation (4) ignores the time needed to construct this testing keystream;
- Assumption #2: Each table-search is viewed as a single 'operation' (i.e., D table-searches require D time units). In practice, however, search-algorithms may require different numbers of operations;

- Assumption #3: False alarms are expected to be rare enough that the time required to perform a test-decryption on each 'hit' is ignored.

How might (4) look without these assumptions? First, since each byte of the D-bit testing keystream is simply 1 byte of X' XOR'd to 1 byte of Y', constructing the testing keystream would add another $D/8 = N_{X'}/2$ operations to the time complexity. Next, each table-search would add $\sim \log(M)$ operations, assuming binary-type sort/search algorithms. Finally, each Q-bit test-decryption would add Q KSG-update operations. With these modifications, a more accurate representation of the time complexity of a standard BG-attack would be

$$T_{BG} = N_{X'} \cdot \frac{1}{2} + N_{searches} \cdot \log(M) + N_{decrypts} \cdot Q,$$

where $N_{searches}$ and $N_{decrypts}$ are the number of table-searches and test-decryptions. This simplifies to

$$T_{BG} \approx D\left(\frac{1}{8} + \frac{\log(M)}{2}\right) + Q$$

under the usual assumptions that half the table needs to be searched (i.e., $N_{searches} = D/2$) and that false alarms are rare (i.e., $N_{decrypts} \approx 1$). Since both $\log(M)$ and Q are typically $\ll D$, the time-complexity is still $\mathcal{O}(D)$, and so, (4) is not a bad approximation.

But what if this was not the case? Specifically, what if the number of test-decryptions or operations per table-search was *not* small compared to D? As will be discussed next, this more accurately describes what happens during the modified BG-attack.

Modified BG-Attack: Time-Complexity

The time-complexity of the mBG-attack can be measured in different ways. The simplest is based on the idea that the attack uses N_{models} tentative keystreams, each of which requires T_{model} processing operations. In turn, T_{model} depends upon the number of table-searches and the number of test-decryptions

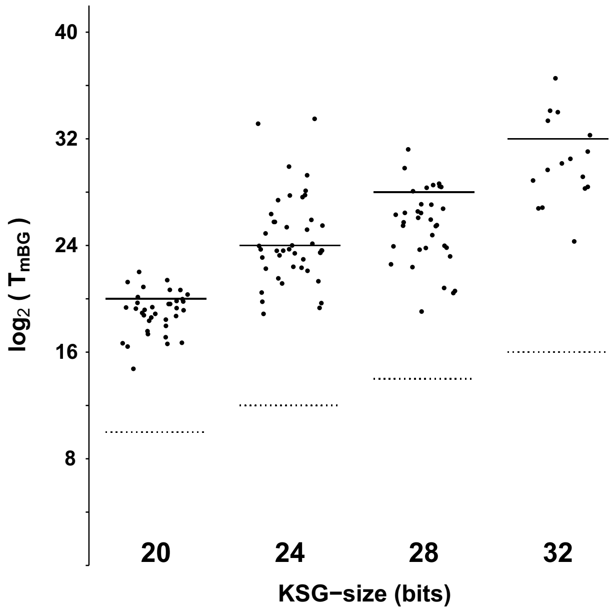


Fig. 3 *Time-complexity of the mBG-attack*. The log-scaled time-complexity is plotted against KSG inner state size ($\log_2(N)$ bits). Dotted line segments mark the $\mathcal{O}(\sqrt{N})$ time-complexity of a traditional BG-attack; solid line segments mark the $\mathcal{O}(N)$ time-complexity of an exhaustive state-space search. T_{mBG} exceeds the time-complexity of a traditional BG-attack, and often approaches or even exceeds that of a brute-force attack. mBG, modified Babbage–Golić; KSG, keystream generator

$$\begin{aligned} T_{mBG} &= N_{models} \times T_{model} \\ &= N_{models} \times (N_{searches} + N_{decrypts}). \end{aligned} \tag{5}$$

A more precise formula for T_{mBG} assigns different weights to each cryptanalytic task. First, a tentative keystream must be constructed from a model with $N_{codewords} = D/4$ codewords—one for each known-plaintext nibble. Essentially, building each nibble of K^T requires a random-number generator call plus several more XOR, AND, and bit-shifts. For convenience, we count all this as four 'operations', so that $N_{codewords} \times 4 = D$ more operations will be used. Next, consider the cost of performing table-searches. Since the modified BG-attack uses a row-by-row search strategy instead of a binary search,[2] this increases the time-complexity by $N_{searches} \cdot M$ operations, where 'operation' means bit-wise multiplication and comparison of two n-bit strings. Finally, the time-complexity must include the cost of doing a test-decryption on all the high-quality hits. Since a Q-bit test-decryption requires $\sim (5.2)Q$ KSG-updates, this adds another $N_{decrypts} \cdot (5.2)Q$ operations to the time-complexity. From these values, a more precise estimate of time-complexity would be

[2] It might be argued that the attacker would be better off using more advanced search/sort algorithms instead of a naive row-by-row search. For two reasons, however, this may not be the case. First, each search can produce multiple hits, a condition which reduces the efficiency of binary-type search algorithms. Second, before each search, all of the prefixes of the table must be bit-wise multiplied by the current verified-sequence fragment. This requires M operations. Thus, while specialized search-algorithms would reduce the number of comparisons from M to $\log(M)$; nevertheless, the entire search process (including the bit-wise multiplications) would need $M + \log(M)$ operations, which is still $\mathcal{O}(M)$.

$$\begin{aligned} T_{mBG} &= N_{models} \times T_{model} \\ &\approx N_{models} \times \big(N_{codewords} \cdot 4 \\ &\quad + N_{searches} \cdot M + N_{decrypts} \cdot (5.2)Q\big) \\ &\approx N_{models} \times \big(D + N_{searches} \cdot M + N_{decrypts} \cdot (5.2)Q\big). \end{aligned} \tag{6}$$

This version of T_{mBG} may easily exceed $\mathcal{O}(\sqrt{N})$. For instance, if $D = M = \sqrt{N}$ and $N_{searches} \approx D$ (as per the traditional BG-attack), then the $N_{searches} \cdot M$ term will have complexity $\mathcal{O}(D \cdot M) = \mathcal{O}(N)$.

Empirical Results

Here, we report the results of many hundreds of mBG-attacks, showing how T_{mBG} depends on parameters controlled by the attacker (i.e., Hamming-weight threshold θ, table size M, and quantity of known-plaintext D) and by the message sender (i.e., KSG-size n).

Most attacks [i.e., most (n, θ, M, D) parameter combinations] were repeated two-to-ten times, each with a different secret key and different sample of known plaintext. For all attacks, the plaintext source was an English-language ASCII text [45]. Four 'toy' KSGs were used: two nonlinear feedback shift registers (NLFSRs) of sizes 20 and 24 bits, and two linear feedback shift registers (LFSRs) of sizes 28 and 32 bits. NLFSR polynomials were from Dubrova's well-known source [20], but because this list stops at n=25, LFSRs were used for the larger sized KSGs. We emphasize that none of these simple KSGs (especially LFSRs!) are intended to be 'secure'. Each one is merely a convenient tool for studying different KSG-sizes.

Effect of n on T_{mBG}

How is time-complexity related to the KSG-size? Figure 3 shows the base-2 logarithm of time-complexity from Eq. (5) vs. KSG-size from 226 mBG-attacks. A small random jitter is included for visual clarity. The horizontal line segments in the figure are not averages or confidence bounds, but rather markers for certain time-complexity cutoffs. The solid line segments mark where $T_{mBG} = N$ (i.e., the time-complexity of a brute-force search of state-space), and the dotted segments mark where $T_{mBG} = \sqrt{N}$ (i.e., the time complexity of a traditional BG-attack against an n-bit stream-cipher system not using PudgyTurtle).

The data suggest a reduced 'scatter' of data-points relative to n (i.e., especially for larger KSGs), which might lead to a false conclusion that T_{mBG} approaches some limiting value for large n. In fact, however, this phenomenon is simply because attacks against larger KSGs took longer, and so fewer of them were performed. For example, an off-the-shelf IBM laptop running research-level (not production-grade) software could easily complete attacks against the 20-bit KSG in minutes or hours, but could take weeks for even a single successful attack against the 32-bit KSG. Thus, we anticipate that the scatter of points at each KSG-size will be similar—provided that one waits long enough to collect the same amount of data for each n.

Several conclusions may be drawn from Fig. 3. First and as expected, the attack's time-complexity increases as the inner state size of the KSG increases. Next, T_{mBG} always exceeds $\mathcal{O}(\sqrt{N})$, suggesting that the PudgyTurtle process adds significant time-complexity compared to a trade-off attack against an analogous (n-bit) system not using PudgyTurtle. Some T_{mBG} even exceed the time-complexity of an exhaustive search.[3] Finally, the scatter of T_{mBG} values for each n suggests that it may be possible to speed up the attack by choosing the right parameters.

Using this same data, Table 5 compares the two ways to measure time-complexity, with the left column using the simpler formula (5) and the right column the more precise, weighted estimate (6). To facilitate comparison among multiple KSG-sizes, this table reports the 'normalized logarithm' of time-complexity

Table 5 Two ways to measure time-complexity

	$\eta(T_{mBG}) = \log_2(T_{mBG})/n$	
	'Simple'	'Weighted'
	Equation (5)	Equation (6)
Mean [95% CI]	1.0813 [1.06, 1.10]	1.5385 [1.51, 1.57]
Minimum	0.68	1.14
Maximum	1.52	2.02
% with $\eta(T_{mBG}) < 1$	33	0

The normalized logarithmic time-complexity $\eta(T_{mBG}) = \log_2(T_{mBG})/n$ is shown for mBG-attacks against KSGs of size 20, 24, 28, and 32 bits. The left-hand column uses the simpler time-complexity measure (5); the right-hand column uses the more precise one (6), which weights each term differently. For both measures of time-complexity, the average, 95% confidence interval, range, and the percentage of attacks with $\eta(T_{mBG}) < 1$ are given. Note that $\eta(T_{mBG})$ of the 'weighted' estimate on average exceeds that of the simpler estimate, and is always > 1. CI: confidence interval; mBG, modified Babbage-Golić; KSG: keystream generator s

[3] We emphasize that $T_{mBG} > \mathcal{O}(N)$ *does not* imply that PudgyTurtle is 'more secure' against brute-force cryptanalysis than other cryptosystems! Rather, these large values illustrate that the mBG-attack is not a particularly efficient way to run a brute-force search (e.g., some KSG-states are tried multiple times, and not every KSG-state is checked at each bit-offset within the tentative keystream). A more reasonable interpretation would be to view attacks for which $T_{mBG} > N$ as cases where brute-force cryptanalysis would be a reasonable alternative to the mBG-attack, with respect to time requirements.

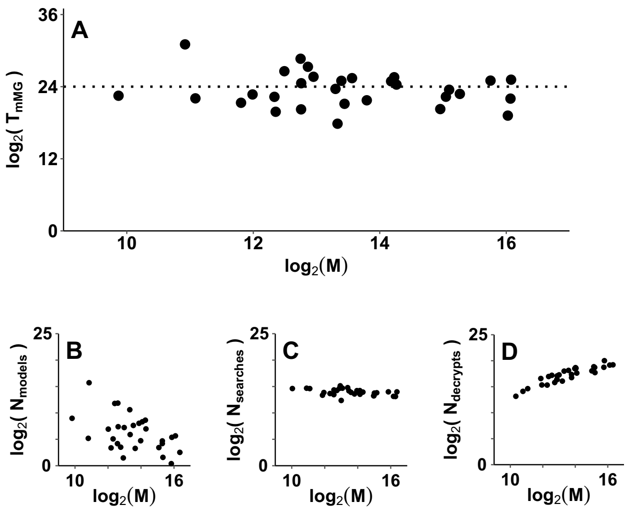


Fig. 4 Effect of table size on time-complexity. **A** Time-complexity T_{mBG} is plotted against table size M (both log-scaled), for attacks against a 24-bit KSG, with θ = 6 and D = 4096. Panel B shows $\log_2(N_{models})$; **C** shows $\log_2(N_{searches})$; and **D** shows $\log_2(N_{decrypts})$, all along the same X-axis as the top panel. As M grows larger, fewer models are needed, but more test-decryptions are performed on each one—opposing trends which leave T_{mBG} itself relatively unaffected. mBG, modified Babbage–Golić; KSG: keystream generator

$$\eta(T_{mBG}) = \log_2(T_{mBG})\ /\ n.$$

Thus, $\eta(T_{mBG}) = 1/2$ represents the $\mathcal{O}(\sqrt{N})$ complexity of a traditional BG-attack; and $\eta(T_{mBG}) = 1$ the $\mathcal{O}(N)$ complexity of a brute-force attack. The mean, 95% confidence interval, range, and % of attacks in which $\eta(T_{mBG})$ falls below 1 are all reported, Notice that the weighted estimate tends to exceed the simpler one, and always exceeds 1.

In the rest of this manuscript, (5) will be used to measure time-complexity. Since this simpler formula tends to result in smaller values than (6), this gives the most favorable assumptions to the attacker, and has the added benefit of making some upcoming analysis (e.g., "Estimated Time-Complexity") more straightforward.

Effect of M on T_{mBG}

Figure 4A shows $\log_2(T_{mBG})$ vs. $\log_2(M)$ for attacks in which table size M was varied (from as low as 1024 to as high as 65,536 rows), while keeping KSG-size n fixed at 24 bits, D at 4096 bits and θ at 6. Intuitively, it would seem that bigger tables should lead to faster attacks: since each tentative keystream fragment would get compared against more rows, more high-quality hits would occur, and each model would thus be more likely to succeed. Interestingly, this is not the case: T_{mBG} is not greatly affected by changes in M.

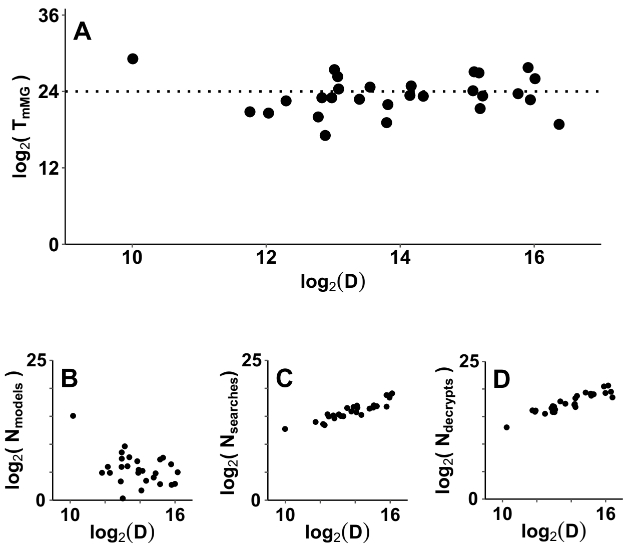


Fig. 5 Effect of data on time-complexity. **A** Time-complexity $\log_2(T_{mBG})$ is plotted against the quantity of known plaintext data, $\log_2(D)$, for attacks against an n = 24-bit KSG with θ = 6 and M = 4096 rows. Panel **B** shows $\log_2(N_{models})$; **C** shows $\log_2(N_{searches})$; and **D** shows $\log_2(N_{decrypts})$, all plotted against the same X-axis as in the top panel. The time-complexity is not significantly affected by changes in D, consistent with the observation that as D increases, N_{models} and its co-factor in the time-complexity (i.e., $N_{searches} + N_{decrypts}$) change in opposite directions. KSG: keystream generator; mBG, modified Babbage–Golić

How might this be explained? Consider the quantities that contribute to time-complexity: N_{models}, $N_{searches}$, and $N_{decrypts}$. First, since larger tables allow for more comparisons against a fixed amount of tentative keystream, the number of models drops as M increases (Fig. 4B). Next, since table size does not affect the *number* of searches per model (which just depends on D, n, and θ), $N_{searches}$ remains constant (Fig. 4C). Finally, since bigger tables mean that each search can yield more high-quality hits (i.e., a single tentative keystream fragment can hit more rows of a bigger table), $N_{decrypts}$ increases (Fig. 4D). The net effect of reducing N_{models} while increasing $N_{decrypts}$ makes time-complexity relatively insensitive to changes in table size. Larger tables mean more high-quality hits and test-decryptions among fewer models, while smaller tables mean fewer decryptions among more models.

Effect of D on T_{mBG}

How does the amount of known-plaintext affect time-complexity? To study this, we mounted a set of attacks in which D was varied between 1024 and 65,536 bits, while fixing the other parameters at $n = 24$, $M = 4096$, and $\theta = 6$. Results are shown in Fig. 5. Changing D has little effect on time-complexity (Fig. 5A). As the amount of data increases, fewer models are needed (Fig. 5B), but more time is spent on each model—for both searches (Fig. 5C)

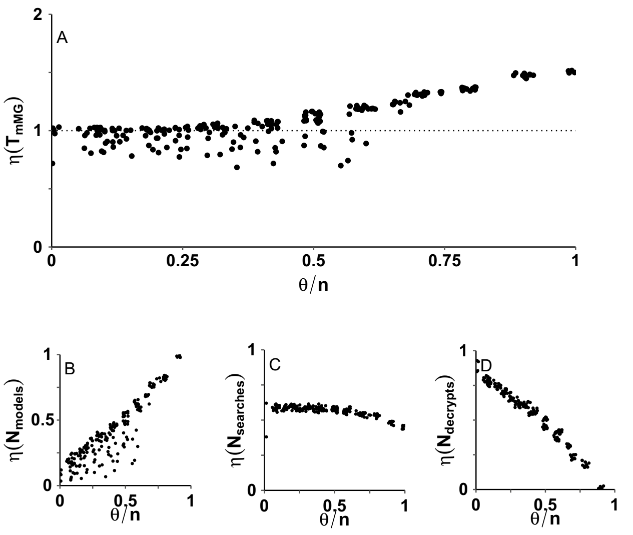


Fig. 6 Effect of Hamming-weight threshold on time-complexity. **A** Normalized logarithmic time-complexity, $\eta(T_{mBG}) = \log_2(T_{mBG})/n$, is plotted against normalized Hamming-weight threshold θ/n for attacks against n = 20-, 24-, 28-, and 32-bit KSGs, with $D = M = \sqrt{N}$ and various $2 \le \theta \le n-2$. The dotted line represents the time-complexity of a brute-force attack. Panels **BD** show the normalized logarithm of N_{models}, $N_{searches}$, and $N_{decrypts}$, respectively. Notice that time-complexity trends upwards with larger θ, suggesting that the rise in N_{models} outweighs the fall in $N_{searches}$ and $N_{decrypts}$. KSG, keystream generator mBG, modified Babbage–Golić

and test-decryptions (Fig. 5D). Intuitively, larger D means each model produces a longer tentative keystream; more table-searches, high-quality hits, and test-decryptions; but that correspondingly fewer models will be necessary. Again, these two opposing trends make the product, T_{mBG}, less sensitive to changes in D.

Effect of θ on T_{mBG}

Previously, it was hypothesized that extremes of θ would lead to slower, less-efficient attacks [3]. Smaller values (e.g., $\theta \le n/4$) would waste too much time on false alarms—since each tentative keystream, with so many unknown bits, could generate so many spurious hits. Larger values (e.g., $\theta \ge 3n/4$) would expend too much time-constructing models—since each one would lead to few (if any) high-quality hits. Either way, T_{mBG} would increase. We therefore suggested that choosing mid-range $\theta/n \in [\frac{2}{3}, \frac{3}{4}]$ (the middle of its range) would balance these two factors, allowing successful attacks in a reasonable amount of time. Here, we vary θ over a wider range. Interestingly, (see below) this original intuition was wrong: although mBG-attacks with smaller θ do indeed generate more test-decryptions, the time-complexity of these attacks still falls below that of attacks with larger θ.

Figure 6A shows the normalized-logarithmic time-complexity, as a function of θ/n for attacks against 20-, 24-, 28-, and 32-bit KSGs, using various $\theta \in \{2, 4, 6, \dots, n-2\}$. The dotted line at $\eta(T_{mBG}) = 1$ represents $\mathcal{O}(N)$ time-complexity. Figure 6B shows $\eta(N_{models})$; 6C shows $\eta(N_{searches})$, and 6D shows $\eta(N_{decrypts})$, all plotted against θ/n as well.

The time-complexity exhibits an upward trend, especially above $\theta/n = 0.5$. That is, low-θ attacks actually take *less* time than high-θ attacks. Larger θ means that fewer fragments exceed threshold and are chosen for a table-search (Fig. 6C), with correspondingly fewer test-decryptions (Fig. 6D), which in turn means that more models are necessary for the attack to succeed (Fig. 6B). Just as in Figs. 4 and 5, there is an opposing trend between N_{models} and the time-per-model ($N_{searches} + N_{decrypts}$). However, in this case, the net result differs: there is less of a time-penalty for lowering θ (using fewer models, each with more false alarms) than for raising θ (using more models, each with few false alarms).

Parameterizing the Time-Complexity

Here, we develop an expression for T_{mBG} in terms of system parameters (n, θ, M, and D), rather than the experimentally derived quantities N_{models}, $N_{searches}$, and $N_{decrypts}$. To do so, these measured quantities will be expressed in terms of an underlying probability distribution which represents the number of 'known' bits in each n-bit fragment of tentative keystream.

The number of known bits in fragment $K^T(b)$ is just the Hamming-weight of its corresponding verified-sequence fragment $V(b)$

$$\# : \{\text{ known bits in } K^T(b)\} = h(V(b)),\ \forall\, b \in \{1, 2, \dots, \ell\}.$$

Letting discrete random-variable $\mathcal{H}$ represent the Hamming-weight of an n-bit fragment of verified sequence, its probability mass function (p.m.f.) supported on $a \in \{0, 1, 2, \dots, n\}$ is written as

$$p_{\mathcal{H}}(a) \equiv \Pr\{\mathcal{H} = a\}.$$

Using this p.m.f., we can compute three important probabilities: the probability that an n-bit fragment of K^T will be chosen for a table-search; the probability of that a table-search will produce a high-quality hit; and the probability that a high-quality hit will lead to a successful test-decryption. These three probabilities will then be used to estimate $N_{searches}$, $N_{decrypts}$, and N_{models}, which then lead to an estimate of the time-complexity itself.

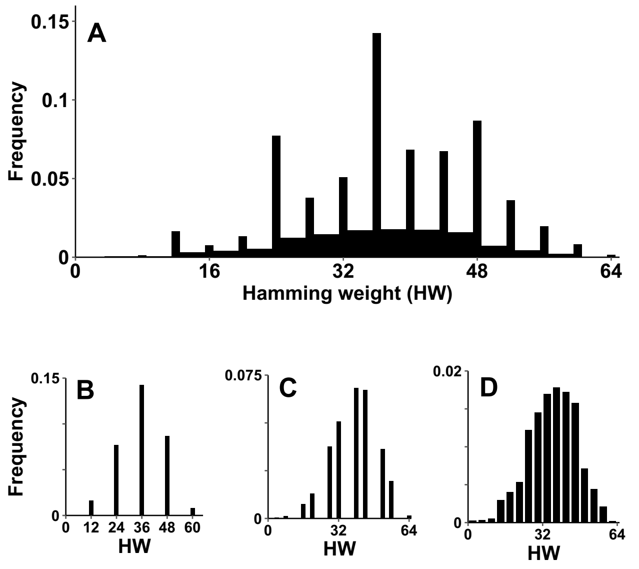


Fig. 7 Hamming-weight distribution. Panel **A** shows $p_{\widehat{\mathcal{H}}}$, the frequency distribution of Hamming-weights of $n = 64$-bit fragments of a simulated 5,200,000-bit verified sequence. The unusual, multi-peaked shape is composed of three sub-distributions: one at Hamming-weight multiples of 12 (**B**); one at multiples of 4, but not 12 (**C**); and one at multiples of 1, 2, and 3 (mod 4), whose thicker looking elements are actually three closely spaced bars of similar amplitude (**D**). Notice that each of these distributions is 'bell-shaped'; that each is slightly skewed with maxima at $> n/2 = 32$; and that amplitudes are highest for Hamming-weights that are multiples of 12, mid-range for multiples of four, and lowest for weights of 1, 2, or 3 (mod 4)

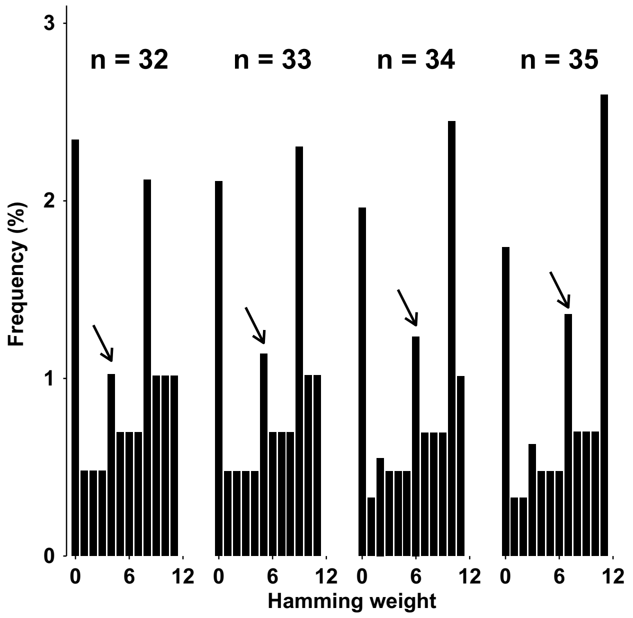


Fig. 8 *Hamming-weight distribution for different KSG-sizes.* The lower tails (Hamming-weights < 12) of Hamming-weight distributions from simulated 5,200,000-bit verified sequences for n = 32, 33, 34, and 35. Each distribution has a similar shape, but its peaks are right-shifted by n (mod 4). In the leftmost distribution ($n = 32 = 0$ modulo 4) for example, the arrow shows a peak at Hamming-weight of 4. As n increases to 33, 34, and 35, this peak is seen at 5, 6, and 7, respectively. KSG, keystream generator

Hamming-Weight Distribution

We begin by describing the distribution of Hamming-weights of n-bit fragments of V. While this distribution can be obtained by mining data from mBG-attacks, there is also a 'shortcut' for getting these data: simulation. The verified sequence simply marks the known/unknown status of each bit of K^T. Filling in the actual bits of a tentative keystream requires an mBG-attack (i.e., information about X' and Y'), but simply marking each of its bits as 'known' or 'unknown' just requires the model's failure counters. Since these failure counters are randomly selected during an mBG-attack, so too can they be randomly selected 'in isolation', thus providing the basis for a simulated verified sequence, $\widehat{V}$. From this, frequency-counting then provides the simulated Hamming-weight distribution $p_{\widehat{\mathcal{H}}}$

$$p_{\widehat{\mathcal{H}}}(a) = \Pr\{h(\widehat{V}(b)) = a\} = \frac{1}{|\widehat{V}|} \cdot \sum_{b=1}^{|\widehat{V}|} \delta(h(\widehat{V}(b)) - a),$$

where the delta function is

$$\delta(u) = \begin{cases} 1 & \text{if}\, u = 0 \\ 0 & \text{if}\, u \neq 0. \end{cases}$$

There are two ways to simulate a verified sequence. One method is to randomly choose a geometrically distributed failure counter $f \in \{0, 1, 2, \dots, 31\}$; construct a 'word' of format `0xFF0...0F` with f zeros; concatenate this word to the existing sequence; and repeat this procedure until some desired total length is reached. The other method is to use a Markov process (see below), whose two states (`0x0` and `0xFFF`) have transition probabilities that reflect the plaintext-to-keystream matching process during PudgyTurtle encoding.

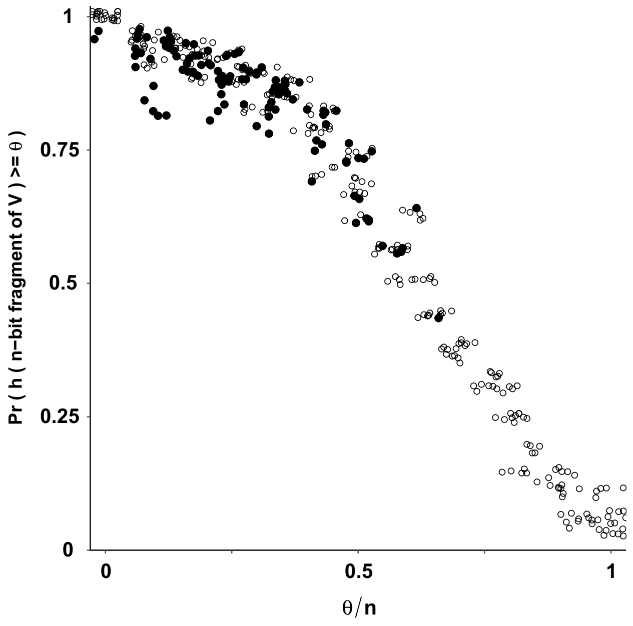
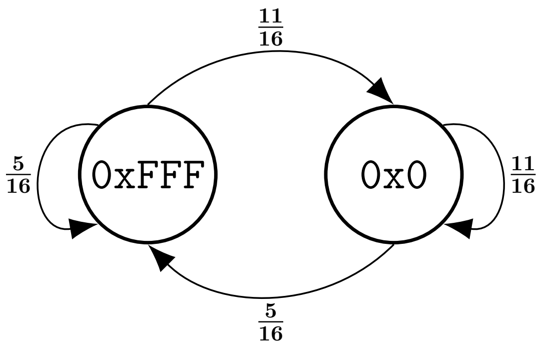


Fig. 9 *Probability of θ or more known bits.* The probability that an n-bit tentative-keystream fragment has $\geq \theta$ known bits is plotted against the normalized Hamming-weight threshold θ/n. Data are shown from simulations (∘) and actual mBG-attacks (•). Simulated probabilities were calculated from 5,200,000-bit verified sequences, to which values of $n \in \{20, 24, 28, 32\}$ and $0 \leq \theta \leq n$ were assigned. Measured probabilities were obtained by dividing the observed number of table-searches by the tentative-keystream length, using only those attacks in which at least four models were available to average. mBG, modified Babbage–Golić



The first method does not produce any overflow events, just like the tentative keystreams used in an actual mBG-attack. The second method not only does allow overflows (i.e., words with ≥ 32 zero-nibbles) but also fails to mark them (e.g., with an extra `0xFF` codeword). Since this anomaly might slightly affect the resulting p.m.f., we will therefore use the first simulation method.

One example of a simulated Hamming-weight distribution (for $n = 64$) is shown in Fig. 7A. Its unusual, characteristic multi-peak appearance seems to be composed of three sub-distributions. The highest-amplitude peaks correspond to Hamming-weights that are multiples of 12 (Fig. 7B); the next-highest peaks to Hamming-weights that are multiples of 4, but not 12 (Fig. 7C); and the lowest-amplitude peaks to Hamming-weights that are *not* multiples of four (Fig. 7D). While this distribution is easy to simulate and its shape is straightforward to describe, an analytical expression for $p_{\mathcal{H}}$ remains an open question.

When KSG-size n is not a multiple of four, the Hamming-weight distribution has the same general shape, but its peaks are shifted rightwards by n (mod 4). Figure 8 shows close-up views of the left tails (first 12 values) of simulated Hamming-weight distributions for $n = 32$, 33, 34, and 35. Notice that each time n increases by one, the peaks shift rightwards as well. For instance, the peak at 4 (shown by the arrow) moves along the X-axis to 5, 6, and 7.

$N_{Searches}$

The next step is to use the observed $p_{\mathcal{H}}$ or simulated $p_{\widehat{\mathcal{H}}}$ to parameterize each of the three components of time-complexity in Eq. (5).

For the number of table-searches, what is needed is a probabilistic expression for the rate at which n-bit fragments of tentative keystream contain θ or more known bits

$$P_A(n, \theta) = \Pr\{\mathcal{H} \geq \theta\} = \sum_{j=\theta}^{n} p_{\mathcal{H}}(j),$$

where A stands for 'above threshold'. Averaging over the whole tentative keystream, we get

$$\widehat{N}_{searches} = \ell \cdot P_A(n, \theta) \approx (5.2)D \cdot \sum_{j=\theta}^{n} p_{\mathcal{H}}(j). \qquad (7)$$

Figure 9 shows $P_A(n, \theta)$ as a function of θ/n, from simulated verified sequences (∘) and actual mBG-attacks (•). Simulations involved 5,200,000-bit $\widehat{V}$ sequences, each representing the amount of keystream required to encrypt ~ 1 million bits. For each $\widehat{V}$, values of $n \in \{20, 24, 28, 32\}$ and $\theta \in \{0, 1, 2, \ldots, n\}$ were assigned; $P_A(n, \theta)$ was calculated; and the process repeated with different random seeds to obtain an average value. For mBG-attacks, the probability was obtained by dividing the measured number of table-searches by $|K^T|$, and averaging over N_{models} repetitions. This figure demonstrates the close agreement between simulated and measured values.

$N_{Decrypts}$

Since a test-decryption is done following every high-quality hit, and since high-quality hits are a sub-type of all hits,

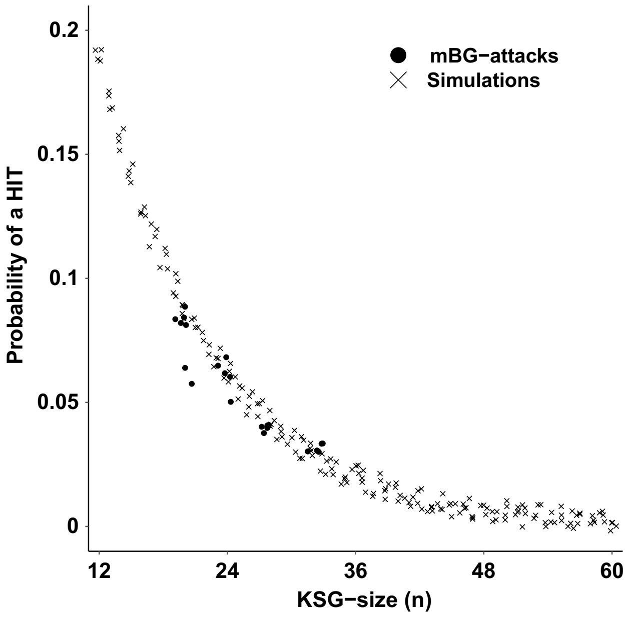


Fig. 10 *Probability of a hit.* The probability $P_{HIT}(n)$ that an n-bit tentative-keystream fragment 'hits' a row of the precomputed table is shown against KSG-size n, using data from actual mBG-attacks (• symbols) and simulations (× symbols). Notice that the $P_{HIT}(n)$ falls exponentially as n increases. KSG: keystream generator; mBG, modified Babbage–Golić

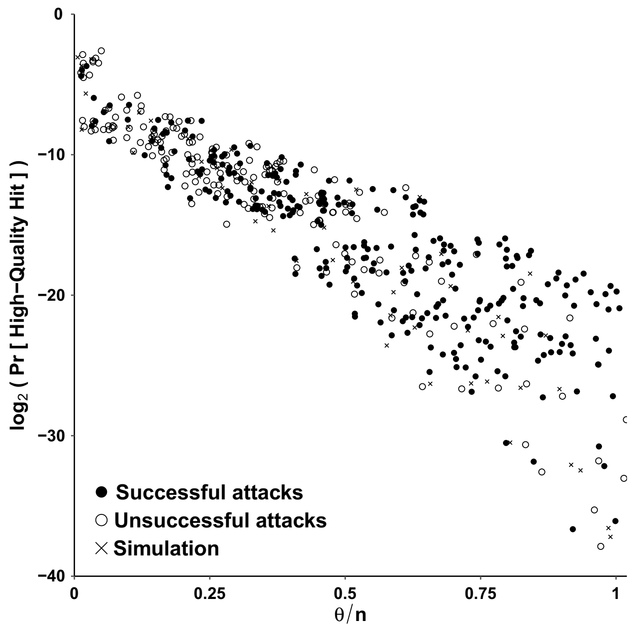


Fig. 11 *Probability of a high-quality hit.* Shown here is the base-2 logarithm of $P_{HQH}(n, \theta)$, plotted against θ/n. Data are from simulations (×), successful attacks (•), and planned unsuccessful attacks (∘). For simulations, the probability comes from the second term in Eq. (9), with $\ell = 5{,}200{,}000$ bits; $n \in \{20, 24, 28, 32\}$; $M = D = \sqrt{N}$, and various $\theta \in \{0, 2, 4, \dots, n\}$). For mBG-attacks, the probability was obtained by computing $N_{decrypts}/(M \times N_{searches})$. Notice the similarity between the measured and simulated data, and also that as θ/n approaches 1, probability estimates become less precise

$N_{decrypts}$ can be discussed by starting with the probability of a hit ('mimic') itself.

Probability of Any Hit

Recall from Sect. 4.2 that a hit between the n-bit tentative-keystream fragment $K^T(b)$ and the n-bit prefix P_h means that $V(b) \otimes K^T(b) = V(b) \otimes P_h$, or equivalently $V(b) \otimes U = 0$, where U is a short-hand for $(K^T(b) \oplus P_h)$. We refer to this equality as the *hit condition*.

Let v be a single bit of verified-sequence fragment $V(b)$. The hit condition will be satisfied trivially whenever $v = 0$. Since prefixes are chosen randomly and uniformly, and since the table is built independently of the model, U can be viewed as a uniformly distributed n-bit vector, so that the hit condition will also be satisfied about half the time when $v = 1$. Since $V(b)$ contains $h(V(b))$ 1's, the per-fragment probability that $K^T(b)$ is a hit is

$$\Pr\{V(b) \otimes U = 0\} = \left(\frac{1}{2}\right)^{h(V(b))}.$$

To obtain the per-keystream probability, we average this quantity over all bit-offsets within the ℓ-bit tentative keystream

$$\begin{aligned}
P_{HIT}(n) &= \left(\frac{1}{\ell}\right) \sum_{b=1}^{\ell} \left(\frac{1}{2}\right)^{h(V(b))} \\
&= \left(\frac{1}{\ell}\right) \left[\left(\frac{1}{2}\right)^{h(V(1))} + \left(\frac{1}{2}\right)^{h(V(2))} + \cdots + \left(\frac{1}{2}\right)^{h(V(\ell))} \right] \\
&= \left(\frac{1}{\ell}\right) \left[(\#_0)\left(\frac{1}{2}\right)^0 + (\#_1)\left(\frac{1}{2}\right)^1 \right. \\
&\quad \left. + (\#_2)\left(\frac{1}{2}\right)^2 + \cdots + (\#_n)\left(\frac{1}{2}\right)^n \right] \\
&= \sum_{j=0}^{n} \frac{(\#_j)}{\ell} \cdot \left(\frac{1}{2}\right)^j,
\end{aligned}$$

where the summation index changes from $b = 1, 2, \dots, \ell$ to $j = 0, 1, \dots, n$ using the notation $(\#_j)$ to represent the number of n-bit fragments of K^T which contain j known bits. Finally, since $(\#)_j/\ell$ is just the probability $p_{\mathcal{H}}(j)$

$$P_{HIT}(n) = \sum_{j=0}^{n} p_{\mathcal{H}}(j) \left(\frac{1}{2}\right)^j.$$

Figure 10 shows $P_{HIT}(n)$ vs KSG-size for data from mBG-attacks (•) and simulations (×). As can be seen, the probability of a hit declines exponentially with KSG-size n, and fits the equation

$$P_{HIT}(n) \approx \left(\frac{12}{20.8}\right) \times 2^{-(0.135152)n}. \quad (8)$$

Note that 12/20.8 is the length of runs of 1-bits in the verified sequence divided by the average length (in bits) of each of its 'words'.

Probability of a High-Quality Hit

In the mBG-attack, hits are either accepted as 'high-quality' (when $K^T(b)$ has $\geq \theta$ known bits), or rejected as 'spurious' (when $K^T(b)$ has $< \theta$ known bits). Thus, $P_{HIT}(n)$ can be split into two terms

$$\begin{aligned} P_{HIT}(n) &= \sum_{j=0}^{\theta-1} p_{\mathcal{H}}(j)\left(\frac{1}{2}\right)^j + \sum_{j=\theta}^{n} p_{\mathcal{H}}(j)\left(\frac{1}{2}\right)^j \\ &= P_{spurious}(n,\theta) + P_{HQH}(n,\theta), \end{aligned} \quad (9)$$

where $P_{spurious}(n,\theta)$ is only defined when $\theta \geq 1$.

From the original ℓ-bit tentative keystream, at most $N_{searches}$ fragments will be chosen as search-targets to compare with the table. However, each of these targets could potentially (e.g., when $\theta = 0$) hit up to M rows in the table. To account for this, the number of test-decryptions per model is defined as

$$\begin{aligned} \widehat{N}_{decrypts} &= \widehat{N}_{searches} \cdot M P_{HQH}(n,\theta) \\ &= \ell P_A(n,\theta) \cdot M \sum_{j=\theta}^{n} P_{HQH}(n,\theta) \\ &\approx (5.2) DM \sum_{j=\theta}^{n} p_{\mathcal{H}}(j) \cdot \sum_{j=\theta}^{n} p_{\mathcal{H}}(j)\left(\frac{1}{2}\right)^j. \end{aligned} \quad (10)$$

Just as simulation offers an easy way to study $\widehat{N}_{searches}$, so too can it be applied to study $\widehat{N}_{decrypts}$ [i.e., $p_{\widehat{\mathcal{H}}}(j)$ can be used interchangeably with $p_{\mathcal{H}}(j)$ in Eq. (10)]. But besides simulation, there is still another short-cut at our disposal: *un*-successful mBG-attacks. Since the number of test-decryptions is a 'per-model' quantity, it is not affected by the ultimate outcome (i.e., success or failure) of cryptanalysis. Thus, rather than waiting for an attack to succeed, useful information can still be obtained by halting an attack after some predetermined number of models have been analyzed, and then averaging $N_{decrypts}$ over this number of models.

Figure 11 compares these different ways to estimate the probability of a high-quality hit. All are $\log_2$-scaled and plotted against θ/n. The • symbols are from successful mBG-attacks; the ∘ symbols are from a series of (planned) unsuccessful mBG-attacks; and the × symbols are from simulations. Attacks used $n = 20, 24, 28$, and 32; $M = D = \sqrt{N}$; various θ from 0 to $(n-2)$; and repetitions with different secret keys and different samples of known-plaintext. The planned unsuccessful attacks were halted once 100 models had been tried—or earlier if the attack succeeded before that point. Either way, the probability of a high-quality hit was calculated by dividing $N_{decrypts}$ by $(N_{searches} \cdot M)$, and averaging over however many models were used. (This was done for successful attacks as well.) Simulations were based on 5,200,000-bit $\widehat{V}$-sequences, with $P_{HQH}(n,\theta)$ calculated using the second summation in (9).

Figure 11 illustrates several important points. First (and as expected), the three estimates of $P_{HQH}(n,\theta)$ are similar. Next, as θ increases, the probability of high-quality hit decreases. Two factors explain this trend: the number of n-bit fragments of K^T chosen to undergo a table-search becomes smaller; and also the chance that one of these fragments matches a prefix diminishes (i.e., this probability falls as the number of known bits per-fragment, and θ, increase). Finally, the three methods for determining $P_{HQH}(n,\theta)$ all become less precise as θ approaches n. In this region of parameter space, relatively few fragments within each K^T possess $\geq \theta$ known bits. In essence, the growing uncertainty as $\theta \to n$ comes from fewer and fewer data-points being included in each probability estimate.

N_{Models}

The mBG-attack analyzes model after model until one tentative keystream finally produces a valid hit. N_{models} is the number of keystreams needed to make this happen. In some sense, then, N_{models} is inversely related to the 'correct test-decryption' probability.

To review, suppose that fragment $K^T(b)$ makes a high-quality hit with prefix P_h (paired with KSG-state S_h in the precomputed table). This newly discovered state is first adjusted ($S_h \to S''_h$), and then used to produce 'regenerated' keystream $K^R = \mathrm{KSG}[S''_h]$. If a test-decryption using K^R correctly matches Q bits of known-plaintext, then K^R must be identical to part of the actual keystream (K^A) used to encrypt the message, and the attack succeeds. Consider several scenarios for the likelihood of a correct test-decryption, based on the number of known bits in $K^T(b)$:

- If none of its bits are known ($h(V(b)) = 0$), then $K^T(b)$ would mimic *all* prefixes in the table; K^R would not necessarily bear any relationship to $K^T(b)$ or K^A; and a correct test-decryption would be unlikely. This could happen during a mBG-attack with $\theta = 0$.
- If all bits of $K^T(b)$ are known ($h(V(b)) = n$), then K^R would usually reproduce K^A, and a correct test-decryption would be likely. This could happen during a 'tra-

ditional' BG-attack (in which false alarms are rare), or during a modified BG-attack with $\theta = n$.

– What if all of $K^T(b)$'s bits are known except for one? For example, consider an mBG-attack with $\theta = n - 1$, during which a high-quality hit is produced by a K^T-fragment for which $h(V(b)) = n - 1$.

Before comparison, P_h and $K^T(b)$ are both multiplied by a vector ($V(b)$) containing $n - 1$ ones and a single zero. This causes the prefix to only have a 50% chance of being linked with the desired KSG-state. In other words, KSG[S_h] always faithfully reproduces P_h, but P_h may not actually equal $K^T(b)$. After all, S_h was obtained after comparing $[P_h \otimes V(b)]$ and $[K^T(b) \otimes V(b)]$, *not* after directly comparing P_h and $K^T(b)$. If the single 0-bit in $V(b)$ causes $K^T(b)$ to be off from the original keystream, then K^R will not match K^A and the test-decryption will be wrong. Since 0- and 1-bits in P_h occur with equal probability, only 50% of the high-quality hits of this type will yield a correct test-decryption.

Generalizing this idea, a correct test-decryption becomes 50% less likely with each unknown bit of $K^T(b)$, so that

$$\left(\frac{1}{2}\right)^{n-h(V(b))}$$

is the probability that a high-quality hit with $K^T(b)$ leads to a correct test-decryption.

Notice that the above is a conditional probability, depending upon $K^T(b)$ being a high-quality hit in the first place. The unconditional probability of interest is

$$\begin{aligned}
P_C(b) &= \Pr\{K^T(b) \rightarrow \text{ correct test-decryption } \cap K^T(b) \text{ is a HQH}\}\\
&= \Pr\{K^T(b) \rightarrow \text{ correct } \mid K^T(b) \text{ is HQH}\} \times \Pr\{K^T(b) \text{ is HQH}\}\\
&= \left(\frac{1}{2}\right)^{n-h(V(b))} \times \Pr\{K^T(b) \text{ is a HQH}\}\\
&= \left(\frac{1}{2}\right)^{n-h(V(b))} \times \Pr\{K^T(b) \text{ is a hit } \cap K^T(b) \text{ has } \geq \theta \text{ known bits}\}\\
&= \left(\frac{1}{2}\right)^{n-h(V(b))} \times \Pr\{K^T(b) \text{ is a hit } \cap h(V(b)) \geq \theta\}\\
&= \left(\frac{1}{2}\right)^{n-h(V(b))} \times \Pr\{K^T(b) \text{ is a hit } \mid h(V(b)) \geq \theta\} \times \Pr\{h(V(b)) \geq \theta\}\\
&= \left(\frac{1}{2}\right)^{n-h(V(b))} \times \left(\frac{1}{2}\right)^{h(V(b))} \times \text{STEP}[h(V(b)) - \theta]\\
&= \text{STEP}[h(V(b)) - \theta] \,/\, N,
\end{aligned}$$

where C stands for 'correct' and STEP is the right-continuous Heaviside function, STEP$[x] = 1$ (if $x \geq 0$) and 0 (if $x < 0$).

To obtain the per-model (not per-fragment) probability, this quantity is averaged all tentative-keystream fragments

$$P_{correct}(n, \theta) = \left(\frac{1}{\ell}\right) \sum_{b=1}^{\ell} P_C(b) = \frac{P_A(n, \theta)}{N}.$$

Since each model will on average produce $\widehat{N}_{decrypts}$ test-decryptions, the number of correct test-decryptions per model is

$$N_{correct}(n, \theta) = \widehat{N}_{decrypts} \cdot P_{correct}(n, \theta),$$

so that the average number of models required for a successful attack (i.e., the number of models per correct test-decryption) will be

$$\begin{aligned}
\widehat{N}_{models} &= \frac{1}{N_{correct}(n, \theta)}\\
&= \frac{1}{(\ell M) P_A(n, \theta) P_{HQH}(n, \theta) \cdot P_A(n, \theta)/N} \qquad (11)\\
&\approx \frac{N}{(5.2) DM \cdot P_{HQH}(n, \theta) \cdot P_A^2(n, \theta)}.
\end{aligned}$$

Estimated Time-Complexity

Using (7), (10), and (11) respectively for $\widehat{N}_{searches}$, $\widehat{N}_{decrypts}$, and $\widehat{N}_{models}$, the time-complexity can be estimated as

$$\widehat{T}_{mBG} = \widehat{N}_{models} \times (\widehat{N}_{searches} + \widehat{N}_{decrypts}),$$

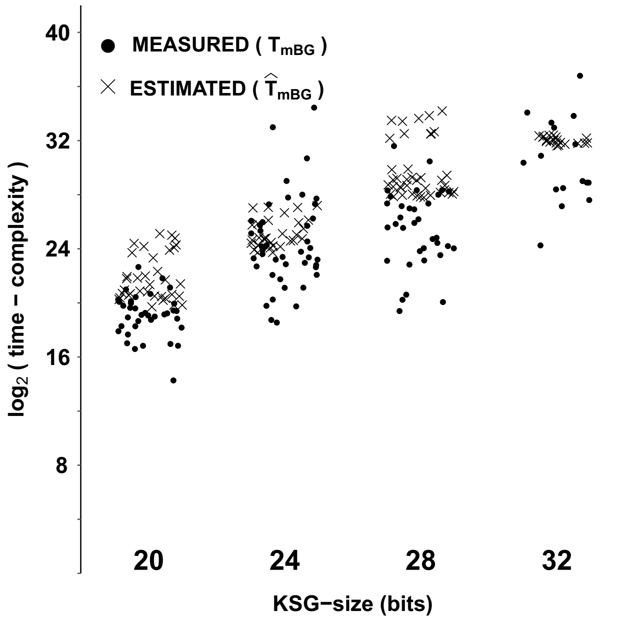


Fig. 12 *Time-complexity: measured vs. estimated.* The logarithmic time-complexity is plotted against KSG-size for two groups of data. The • symbols are empirically measured T_{mBG} values, collected from ~ 130 successful mBG-attacks. The × symbols are estimated $\widehat{T}_{mBG}$ values, obtained from 5,200,000-bit simulated verified sequences with the same (n, θ)-values as the empirical data. Notice that $\widehat{T}_{mBG}$ over-estimates the measured data, most apparently when $n = 20$ and 28. mBG, modified Babbage–Golić; KSG: keystream generator

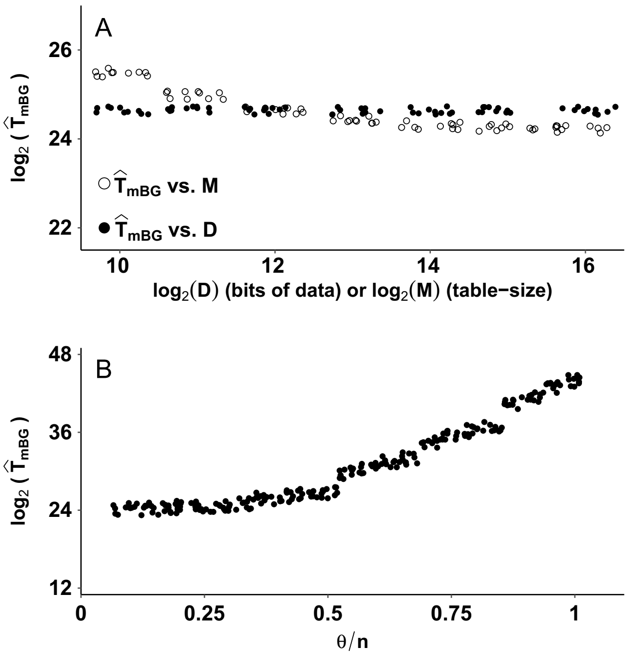


Fig. 13 Estimated time-complexity. **A** Estimated time-complexity $\widehat{T}_{mBG}$ (for n=24 and θ=6) is shown against a $\log_2$-scaled X-axis, which represents either D (the number of known-plaintext bits, shown as • symbols) or M (the number of rows in the table, shown as ∘ symbols). The estimated time-complexity does not change substantially with either D or M. **B** The $\log_2$-scaled estimated time-complexity is plotted against θ/n for $n = 24$, $M = D = 4096$, and various θ. Time-complexity increases with θ, a trend that becomes more noticeable once θ/n exceeds 0.5. mBG, modified Babbage–Golić; KSG: keystream generator

which can be expanded to

$$\widehat{T}_{mBG} = \left(\frac{N}{\ell M \cdot P_A^2(n,\theta) P_{HQH}(n,\theta)} \right) \times [\, \ell \cdot P_A(n,\theta) + (\ell M) \cdot P_A(n,\theta) P_{HQH}(n,\theta)\,] \quad (12)$$
$$= \frac{N}{P_A(n,\theta)} \left(1 + \frac{1}{M P_{HQH}(n,\theta)} \right).$$

Since $0 \le P_A(n,\theta) \le 1$, this estimated time-complexity will be $\ge N$, whereas measured values, like those in Fig. 3, need not always be this large (see "Comparing $\widehat{T}_{mBG}$ and T_{mBG}" for details). To emphasize, however, $\widehat{T}_{mBG} > N$ simply suggests that the mBG-attack would perform no better than brute-force—not that PudgyTurtle is somehow providing security 'beyond' $\mathcal{O}(N)$ complexity.

How well does this estimate reproduce results obtained from actual attacks, and how can $\widehat{T}_{mBG}$ be used to further our understanding of the mBG-attack? Figures 12 and 13 provide some confirmatory findings, while Table 6 and Fig. 14 use $\widehat{T}_{mBG}$ to gain some useful insights.

Table 6 *Time-complexity for 'extreme' θ*

θ	$\widehat{N}_{models}$	Time-per-model	$\widehat{T}_{mBG}$
0	3.14	5.36e+06	1.68e+07
24	1.09e+10	1.42e+03	1.55e+13

Shown here are mBG-attack complexity measures based on simulated verified sequences, assuming a 24-bit KSG, $M = D = 4096$, and Hamming-weight thresholds of $\theta = 0$ or $\theta = 24$, all averaged over 20 repetitions. For each θ (Column 1), the estimated number of tentative keystreams needed for a successful attack ($\widehat{N}_{models}$, Column 2); the estimated time-per-model ($\widehat{N}_{searches} + \widehat{N}_{decrypts}$, Column 3); and the estimated time-complexity ($\widehat{T}_{mBG}$, Column 4) are given. Small θ's lead to fewer models which each require lengthy analysis; and vice versa for large θ. However, this tradeoff is not symmetric: the time-complexity is lower when $\theta = 0$ compared to when $\theta = n$

mBG, modified Babbage–Golić, KSG: keystream generator

Comparing $\widehat{T}_{mBG}$ and T_{mBG}

Figure 12 shows T_{mBG} and $\widehat{T}_{mBG}$ as a function of KSG-size. Notice first that the estimated (×) and measured (•) time-complexities are similar, but that the estimates somewhat exceed measured values—especially in the lower (smaller

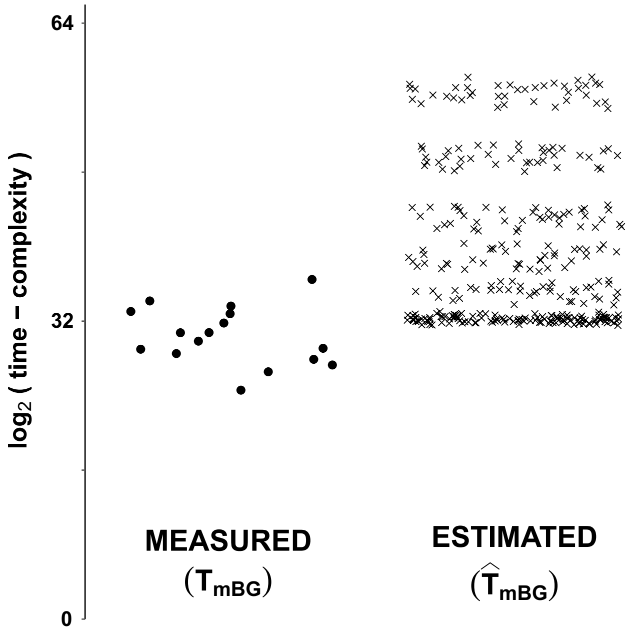


Fig. 14 Time-complexity of attacks against a 32-bit KSG. Measured (left) and estimated (right) time-complexity of attacks against an n = 32-bit KSG, on a $\log_2$ scale. Even for this small KSG (by cryptographic standards), successful mBG-attacks take a long time, and thus, only a few data-points from actual attacks are available, and these are limited to $\theta \le 12$. However, estimating the time-complexity allows visualization of data over the full range of θ. mBG, modified Babbage–Golić; KSG: keystream generator

θ) range. Some of this inaccuracy may be due to implementation-dependent 'endpoint effects'. For example, our software only advances through K^T until the point at which there would no longer be sufficient data for a Q-bit test-decryption. Sometimes, this number may be considerably smaller than the idealized parameter $\ell = |K^T| - n + 1$ (e.g., when $n = 20$, $D = \sqrt{N} = 1024$, and $Q = 96$, this amounts to a difference of $Q/\ell \approx 10\%$). Another contributor to this over-estimation is that attacks halt as soon as a valid hit occurs—which usually happens only part-way through a tentative keystream. If, for example, only one model is needed (again, when θ is small), then $N_{searches}$ will only be tabulated from part of K^T, not from all ℓ of its bits. This would reduce the measured time-per-model (i.e., relatively speaking, inflate the estimate), and could produce similar effects on averaged values even when N_{models} is small but still > 1.

Figure 13 shows how the estimated time-complexity varies with D, M, and θ. Of note, the estimated time-complexity behaves the same way as its empirically measured counterpart: it remains relatively unaffected by changes in D (as in Fig. 5) or changes in M (as in Fig. 4), but increases with θ/n (as in Fig. 6).

Extrapolating with $\hat{T}_{mBG}$

Described next are two usage cases in which the estimated time-complexity—by extending the available parameter range—facilitates more understanding of the mBG-attack.

Extremes of θ

What happens when the mBG-attack is performed with the most 'extreme' Hamming-weight thresholds? At one extreme (when $\theta = 0$), the mBG-attack behaves more like a brute-force approach. Every KSG-state in the table is used repeatedly for test-decryptions starting from each bit-offset within the known plaintext. Few models will be required, but each one will take longer to analyze. At the other extreme (when $\theta = n$), the mBG-attack behaves more like the traditional BG-attack. Each tentative-keystream fragment only rarely (probability $\sim 1/N$) makes a high-quality hit. More models will be required, but the few—if any—high-quality hits in each one can be tested quickly.

These analogies are imperfect. For instance, a real brute-force attack involves trying N KSG-states, whereas the mBG-attack with $\theta = 0$ repeatedly tries the same $M \le N$ states. Similarly, a traditional BG-attack uses every n-bit fragment of known keystream, while the mBG-attack with $\theta = n$ only uses some fragments drawn from many 'hypothetical' (tentative) keystreams.

The time-complexity estimate can be further simplified when $\theta = 0$ and $\theta = n$. The probability of being above threshold is

$$P_A(n,\theta) = \begin{cases} 1 & \text{if } \theta = 0 \\ p_{\mathcal{H}}(n) & \text{if } \theta = n, \end{cases}$$

and the probability of a high-quality hit is

$$P_{HQH}(n,\theta) = \begin{cases} P_{HIT}(n) & \text{if } \theta = 0 \\ p_{\mathcal{H}}(n)/N & \text{if } \theta = n, \end{cases}$$

where the latter comes from the observation that, when θ is zero, all hits are 'high-quality' by definition. Equation (12) can be likewise simplified. When $\theta = 0$, the estimated time-complexity takes its *smallest* value

$$\text{MIN}[\hat{T}_{mBG}] = N\left(1 + \frac{1}{MP_{HIT}(n)}\right) \approx N\left(1 + \frac{20.8}{12} \cdot \frac{N^{0.135152}}{M}\right),$$

where the final approximation uses the curve-fitting equation for the $P_{HIT}(n)$ data in Fig. 10. When $\theta = n$, the time-complexity reaches its *largest* value

$$\text{MAX}[\hat{T}_{mBG}] = \frac{N}{p_{\mathcal{H}}(n)}\left(1 + \frac{N}{Mp_{\mathcal{H}}(n)}\right).$$

Table 6 shows averaged results from 20 simulated extreme-θ mBG-attacks against a 24-bit KSG, 10 with $\theta = 0$ and 10 with $\theta = n$ (i.e., 24). Columns 2 (N_{models}) and 3 (time-per-model) confirm that fewer models each take longer to analyze when $\theta = 0$; and more more models each require less processing-time when $\theta = n$. This inverse relationship between the number of models and the time-per-model might suggest that the mBG-attack performs poorly at both extremes of θ. However, the estimated time-complexity (column 4) demonstrates that this is not the case: the 'trade-off' between N_{models} and time-per-model is not symmetric. Attacks with smaller θ ran faster than attacks with larger θ. The time-penalty for creating models is heavier than the penalty for analyzing them.

Worst-Case Time-Complexity

Another use for the estimated time-complexity is to study mBG-attacks that would otherwise take a prohibitive amount of time. For instance, even though a n=32-bit inner state KSG-size is small by cryptographic standards, nevertheless a modified BG-attack against a 32-bit KSG may take a very long time—especially when θ/n is close to 1. This 'worst-case' cryptanalysis scenario can be better quantified via the estimated time-complexity rather than the measured one.

Figure 14 shows the time-complexity measured from mBG-attacks (• symbols) and estimated from simulated verified sequences (× symbols). While actual attacks only provide data up to $\log_2(T_{mBG}) \approx 32$, the estimation technique broadens the range substantially, to $\log_2(\widehat{T}_{mBG}) \approx 56$.

Although the measured data (left) do not allow any conclusions about the upper bound of T_{mBG}, the expanded view (right) shows it to be $\sim \frac{7}{4} \cdot \log_2(N)$. Again, we emphasize that time-complexities in this large just denote cases in which brute-force would be as reasonable an approach as the mBG-attack. Nevertheless, knowing the magnitude of this upper bound is still helpful: as a target to be lowered, it may help benchmark future improvements in the mBG-attack.

Lightweight Ciphers

The Internet of Things (IoT) is fostering a demand for encryption by devices with significant hardware and software limitations (e.g., 'smart' lightbulbs, RFID tags, and micro-sensor arrays), which in turn has sparked an interest in 'lightweight' ciphers.[4]

While many lightweight block-ciphers have been proposed (including but not limited to KLEIN [25], Simon / Speck [8], KTANTAN [18], PRESENT [12], Piccolo [44], Midori [7], PRINCE [14], LBlock [50], LED [26], and TWINE [32]), there are fewer lightweight stream-ciphers (e.g., Trivium [16], A4 [39], Hummingbird [22], Bean [33], Sprout [1], Plantlet [37], and LIZARD [27]). In part, this relative paucity is due to an oft-cited design criterion that applies to stream (but not block) ciphers: to achieve n-bit security, the inner state size of the stream cipher must be at least $2n$ bits [38]. While this precaution addresses concerns about the Birthday Paradox and TMDT attacks, doubling the inner state size obviously makes it harder to be 'lightweight'.

Recently, two new approaches to lightweight stream cipher design have been proposed. One, discussed by Hamann, Krause, and Meier and instantiated as LIZARD, is based on the so-called 'FP(1)-mode' [27]. Essentially, this involves using the key twice—once (paired with the IV) as an initial state, and once again (via XOR) after adequate mixing of the initial state. This approach can raise the security level of a tradeoff-based key-recovery attack from $\frac{n}{2}$ to $\frac{2n}{3}$ bits. Another approach, proposed by Armknecht and Mikhalev and instantiated as Sprout, is "KSG with Keyed Update Function" (KUF) [1]. This concept involves incorporating part of the secret key into the state-update function, thus reducing its vulnerability to TMDT attacks while still allowing smaller states. Since its introduction, Sprout has been cryptanalyzed in several ways, including not only time–memory tradeoffs [23] but also other attacks involving guess-and-determine strategies, SAT solvers, differential fault analysis, and chosen-IV/related-key determination [6, 28, 34, 36]. A more generalized approach to attacking KUF-based stream-ciphers has also been advanced [31]. Even so, the KUF concept remains an exciting new design approach for lightweight stream-ciphers, and a successor to Sprout (called Plantlet) which addresses various security issues is now available [37].

How does PudgyTurtle fit into the lightweight stream-cipher taxonomy? Since PudgyTurtle is not itself a cipher, neither can it be a 'lightweight cipher'. It can, however, work alongside a lightweight KSG. PudgyTurtle seems to oppose one goal of lightweight cryptography: resource minimization. Its expansion property (i.e., producing about twice as much ciphertext and consuming about five times as much keystream) could be impractical for some IoT applications. Nevertheless, PudgyTurtle shares another goal of lightweight stream-ciphers: resistance to tradeoff attacks. Using PudgyTurtle, an n-bit KSG provides a security level of $> \frac{n}{2}$ against the mBG-attack (i.e., $T_{mBG} > \sqrt{N}$). Thus, while it is wrong to claim that PudgyTurtle inherently turns an existing stream-cipher algorithm into one that is 'lightweight', it is certainly reasonable to continue studying PudgyTurtle within the overall framework of lightweight ciphers. Future work in this context should focus on determining the added hardware cost (e.g., logic blocks and gate equivalents) and performance penalty (e.g., FELICS metrics [19]) of combining the PudgyTurtle process with the existing lightweight stream-ciphers.

Conclusions

We have analyzed a time–memory–data tradeoff attack against PudgyTurtle. This method, the modified-BG (mBG) attack, is based on the well-known work of Babbage and Golić [5, 24]. Using 'toy' keystream generators based on simple feedback shift registers of various sizes, we have shown that the time-complexity of the mBG-attack exceeds $\mathcal{O}(\sqrt{N})$—the bound suggested by the traditional BG-attack against a standard binary-additive stream cipher system. We have also demonstrated how various parameters, including memory, data, and an mBG-specific quantity (Hamming-weight threshold, θ), affect the attack's time-complexity. Specifically, choosing smaller values of θ leads to faster attacks, while changing the table size or amount of data have only limited effects. Finally, we have suggested a way to estimate time-complexity based on a probability distribution which can be simulated. Simulations validate these conclusions, and extend

[4] For example, the US National Institute of Standards and Technology site https://csrc.nist.gov/Projects/lightweight-cryptography/finalists discusses several proposals in this area, and https://cryptolux.org/index.php/Lightweight_Cryptography compares them using several metrics.

them to a wider range of parameter space than could have obtained from actual attacks.

Several questions about the mBG-attack and PudgyTurtle remain as open topics of research:

What is the distribution of the Hamming-weights of n-bit samples of the verified sequence? The $p_{\mathcal{H}}$ distribution has an interesting and unusual shape. An analytical expression for it remains elusive, but could lead to a closed-form solution for the mBG-attack's time-complexity.

Could a better tradeoff attack be designed? Specifically, could some other (non-mBG) TMDT-attack reduce time-complexity down to $\mathcal{O}(\sqrt{N})$? Or could the mBG-attack itself be improved? For example, optimizing the software that does 'model creation' and 'filling-in' could speed up attacks which require many models, thus expanding the attacker's flexibility to choose larger values of θ.

Does the PudgyTurtle process itself introduce vulnerabilities? PudgyTurtle uses keystream differently than a standard binary-additive stream cipher. Instead of enciphering plaintext by XOR'ing it with the keystream, PudgyTurtle uses keystream to encode the plaintext and to encipher the codewords. In some ways, this concept (using the output of an encryption algorithm in a different way) resembles an 'encryption mode'—like ciphertext block chaining [CBC] for block-ciphers. The security of CBC-mode is linked to the security of its underlying block-cipher, but is PudgyTurtle's security similarly linked to the security of its underlying KSG? Or does PudgyTurtle produce its own security problems—in the same way that, for example, poorly implemented CBC-mode may produce vulnerabilities (e.g., padding oracle, non-random, or repeated-IV attacks [46, 48]) despite a strong underlying block-cipher.

Besides TMDT attacks, what other forms of cryptanalysis would succeed against PudgyTurtle? Related to the above, a cipher may be secure against tradeoff attacks yet susceptible to other forms of cryptanalysis. For example, the Advanced Encryption Standard [AES] with large-enough keys is TMDT-resistant, but may be susceptible to side-channel (timing) attacks if not implemented carefully [9, 13, 47]; and stream-ciphers may be tradeoff-resistant but still vulnerable to slid-pair and correlation-based attacks [17, 49]. Since non-TMDT attacks against PudgyTurtle will be inevitable, its apparently good performance against a tradeoff attack is reassuring, but still only one aspect of an overall security evaluation.

Data availability The authors declare that the data supporting the findings of this study are available within the paper and by request. As well, a limited implementation of PudgyTurtle encryption/decryption is available at https://www.github.com/Smaugust/pudgyturtle.

Declarations

Conflict of interest Both authors declare that no funding or grants were received to assist with the preparation of this manuscript, and confirm that no human or animal subjects were involved in this research. Opinions and results contained herein do not imply any official position or endorsement by the Massachusetts General Hospital, Mass General Brigham, McKnight Brain Institute, or the University of Arizona.

References

1. 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. 451–70.
2. August D, Smith A. Pudgyturtle GitHub repository, 2021. https://github.com/smaugust/PudgyTurtle.
3. August DA, Smith AC. Pudgyturtle: using keystream to encode and encrypt. SN Comput Sci. 2020;1(4):Article#226. https://doi.org/10.1007/s42979-020-00221-z
4. August DA, Smith AC. Pudgyturtle: variable-length, keystream-dependent encoding to resist time-memory tradeoff attacks. IACR Cryptology ePrint Archive, Report 2020/838. 2020. https://eprint.iacr.org/2020/838.
5. Babbage S. Improved "exhaustive search" attacks on stream ciphers. In: European convention on security and detection, 1995, Institution of Engineering and Technology. 1995. pp. 161–66.
6. Banik S. Some results on Sprout. In: Biryukov A, Goyal V (eds) 16th international conference on cryptology in India, INDOCRYPT 2015. Lecture notes in computer science INDOCRYPT '15. Springer International Publishing, Berlin; 2015. pp. 124–39.
7. Banik S, Bogdanov A, Isobe T, Shibutani K, Hiwatari H, Akishita T, Regazzoni F. Midori: a block cipher for low energy. In: Iwata T, Cheon JH, editors. Advances in cryptology—ASIACRYPT 2015. Springer, Berlin; 2015. pp. 411–36.
8. Beaulieu R, Treatman-Clark S, Shors D, Weeks B, Smith J, Wingers L. The SIMON and SPECK lightweight block ciphers. In: 2015 52nd ACM/EDAC/IEEE design automation conference (DAC), 2015. pp. 1–6. https://doi.org/10.1145/2744769.2747946.
9. Bernstein DJ. Cache-timing attacks on AES. 2005. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.
10. Biryukov A, Shamir A. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In: Okamoto T, editor. Advances in cryptology—ASIACRYPT 2000. Springer, Berlin; 2000. pp. 1–13.
11. 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. Springer, Berlin; 2001. pp. 1–18.
12. Bogdanov A, Knudsen LR, Leander G, Paar C, Poschmann A, Robshaw MJB, Seurin Y, Vikkelsoe C. PRESENT: an

ultra-lightweight block cipher. In: Paillier P, Verbauwhede I, editors. Cryptographic hardware and embedded systems—CHES 2007. Springer, Berlin; 2007. pp. 450–66.
13. Bonneau J, Mironov I. Cache-collision timing attacks against aes. In: Goubin L, Matsui M, editors. Cryptographic hardware and embedded systems—CHES 2006. Springer, Berlin; 2006. pp. 201–15.
14. Borghoff J, Canteaut A, Güneysu T, Kavun EB, Knezevic M, Knudsen LR, Leander G, Nikov V, Paar C, Rechberger C, Rombouts P, Thomsen SS, Yalçın T. PRINCE—a low-latency block cipher for pervasive computing applications. In: Wang X, Sako K, editors. Advances in cryptology—ASIACRYPT 2012. Springer, Berlin; 2012. pp. 208–25.
15. van den Broek F, Poll E. A comparison of time-memory trade-off attacks on stream ciphers. In: Youssef A, Nitaj A, Hassanien AE, editors. Progress in cryptology—AFRICACRYPT 2013. Springer, Berlin; 2013. pp. 406–23.
16. Cannière CD, Preneel B. Trivium. In: Billet O, Robshaw M (eds) New stream cipher designs. Lecture notes in computer science, vol. 4986. Springer, Berlin; 2008. pp. 244–66.
17. Copeland J, Simpson L. Finding slid pairs for the Plantlet stream cipher. In: Proceedings of the Australasian computer science week multiconference, association for computing machinery, New York, NY, USA, ACSW '20, 2020. https://doi.org/10.1145/3373017.3373024.
18. De Cannière C, Dunkelman O, Knežević M. KATAN and KTANTAN—a family of small and efficient hardware-oriented block ciphers. In: Clavier C, Gaj K, editors. Cryptographic hardware and embedded systems—CHES 2009. Springer, Berlin; 2009. pp. 272–88.
19. Dinu D, Biryukov A, Großschädl J, Khovratovich D, Le Corre Y, Perrin L. FELICS—fair evaluation of lightweight cryptographic systems. 2015. https://www.cryptolux.org/index.php/FELICS. Accessed 2 Oct 2022.
20. Dubrova E. A list of maximum period NLFSRs. IACR Cryptology ePrint Archive, Report 2012/166, 2012. https://eprint.iacr.org/2012/166.
21. 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.
22. Engels DW, Fan X, Gong G, Hu H, Smith EM. Hummingbird: ultra-lightweight cryptography for resource-constrained devices. In: Financial cryptography workshops 2010.
23. Esgin MF, Kara O. Practical cryptanalysis of full Sprout with TMD tradeoff attacks. Cryptology ePrint Archive, Report 2015/289, 2015. https://eprint.iacr.org/2015/289.
24. Golić JD. Cryptanalysis of alleged A5 stream cipher. In: Fumy W, editor. Advances in cryptology—EUROCRYPT '97. Berlin: Springer; 1997. p. 239–55.
25. Gong Z, Nikova S, Law YW. KLEIN: a new family of lightweight block ciphers. In: Juels A, Paar C, editors. RFID: security and privacy. Berlin: Springer; 2012. p. 1–18.
26. Guo J, Peyrin T, Poschmann A, Robshaw M. The LED block cipher. In: Preneel B, Takagi T, editors. Cryptographic hardware and embedded systems—CHES 2011. Berlin: Springer; 2011. p. 326–41.
27. Hamann M, Krause M, Meier W. LIZARD—a lightweight stream cipher for power-constrained devices. IACR Trans Symm Cryptol. 2017;1:45–79.
28. Hao Y. A related-key chosen-IV distinguishing attack on full Sprout stream cipher. Cryptology ePrint Archive, Report 2015/231, 2015. https://ia.cr/2015/231
29. Hellman M. A cryptanalytic time-memory trade-off. IEEE Trans Inf Theor. 1980;26(4):401–6.
30. Kalenderi M, Pnevmatikatos D, Papaefstathiou I, Manifavas C. Breaking the GSM A5/1 cryptography algorithm with rainbow tables and high-end FPGAS. In: 22nd international conference on field programmable logic and applications (FPL); 2012. pp. 747–53.
31. Kara O, Esgin MF. On analysis of lightweight stream ciphers with keyed update. IEEE Trans Comput. 2019;68(1):99–110. https://doi.org/10.1109/TC.2018.2851239.
32. Kobayashi E, Suzaki T, Minematsu K, Morioka S. TWINE: a lightweight block cipher for multiple platforms. In: Selected areas in cryptography, 19th international conference (SAC 2012), vol. 7707. Lecture notes in computer science. Springer, Berlin; 2012. pp. 339–54.
33. Kumar N, Ojha S, Jain K, Lal S. Bean: a lightweight stream cipher. In: Proceedings of the 2nd international conference on security of information and networks, association for computing machinery, New York, NY, USA, SIN '09, 2009. pp. 168–71. https://doi.org/10.1145/1626195.1626238.
34. 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;663–82.
35. Li Z. Optimization of rainbow tables for practically cracking GSM A5/1 based on validated success rate modeling. In: Proceedings of the RSA conference on topics in cryptology—CT-RSA 2016—volume 9610. Springer, Berlin; 2016. pp. 359–77.
36. Maitra S, Sarkar S, Baksi A, Dey P. Key recovery from state information of sprout: application to cryptanalysis and fault attack. IACR cryptology ePrint Archive, Report 2015/236, 2015. https://ia.cr/2015/236.
37. Mikhalev V, Armknecht F, Muller C. On ciphers that continually access the non-volatile key. IACR Trans Symmet Cryptol. 2017;2016:52–79.
38. Mileva A, Dimitrova V, Kara O, Mihaljević MJ. Catalog and illustrative examples of lightweight cryptographic primitives. In: Avoine G, Hernandez-Castro J, editors. Security of ubiquitous computing systems: selected topics. Cham: Springer International Publishing; 2021. p. 21–47.
39. Mohandas NA, Swathi A, R A, Nazar A, Sharath G. A4: a lightweight stream cipher. In: 2020 5th international conference on communication and electronics systems (ICCES), 2020. pp. 573–77. https://doi.org/10.1109/ICCES48766.2020.9138048.
40. 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.
41. Papantonakis P, Pnevmatikatos D, Papaefstathiou I, Manifavas C. Fast, FPGA-based rainbow table creation for attacking encrypted mobile communications. In: 2013 23rd international conference on field programmable logic and applications, 2013. pp. 1–6. https://doi.org/10.1109/FPL.2013.6645525.
42. Rivest RL, Sherman AT. Randomized encryption techniques. In: Chaum D, Rivest RL, Sherman AT (eds) Advances in cryptology: Proceedings of Crypto '82, Springer US, Boston, MA; 1983. pp. 145–63.
43. 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.
44. Shibutani K, Isobe T, Hiwatari H, Mitsuda A, Akishita T, Shirai T. Piccolo: an ultra-lightweight blockcipher. In: Preneel B, Takagi T, editors. Cryptographic hardware and embedded systems—CHES 2011. Berlin: Springer; 2011. p. 342–57.
45. Smith A. An Inquiry into the Nature and Causes of the Wealth of Nations. Project Gutenberg. 2002. http://www.gutenberg.org/ebooks/3300. Retrieved 2 Jan 2021. Urbana, Illinois. 2002.
46. of Standards NI, Technology MD. Recommendations for block cipher modes of operation: Methods and techniques. Tech. Rep. NIST Special Publication SP 800-38A, U.S. Department of Commerce, Washington, D.C. 2001.

47. Tsunoo Y, Saito T, Suzaki T, Shigeri M, Miyauchi H. Cryptanalysis of des implemented on computers with cache. In: Walter CD, Koç ÇK, Paar C, editors. Cryptographic hardware and embedded systems—CHES 2003. Berlin: Springer; 2003. p. 62–76.
48. Vaudenay S. Security flaws induced by CBC padding—applications to SSL, IPSEC, WTLS... In: Proceedings of the international conference on the theory and applications of cryptographic techniques: advances in cryptology, EUROCRYPT '02, 2002. . Springer, Berlin. pp. 534–46.
49. Wang S, Liu M, Lin D, Ma L. Fast correlation attacks on Grain-like small state stream ciphers and cryptanalysis of Plantlet, Fruit-v2 and Fruit-80. IACR Cryptology ePrint Archive, Report 2019/763, 2019. https://ia.cr/2019/763.
50. Wu W, Zhang L. Lblock: a lightweight block cipher. In: Lopez J, Tsudik G, editors. Applied cryptography and network security. Berlin: Springer; 2011. p. 327–44.