Queensland University of Technology
Brisbane Australia

This may be the author's version of a work that was submitted/accepted for publication in the following source:

Beighton, Matthew, Bartlett, Harry, & Simpson, Leonie
(2020)
Security Analysis of Fountain V1.
In *ACSW '20: Proceedings of the Australasian Computer Science Week Multiconference.*
Association for Computing Machinery (ACM), United States of America.

This file was downloaded from: https://eprints.qut.edu.au/201039/

© 2020 Association for Computing Machinery

This work is covered by copyright. Unless the document is being made available under a Creative Commons Licence, you must assume that re-use is limited to personal use and that permission from the copyright owner must be obtained for all other uses. If the document is available under a Creative Commons License (or other specified license) then refer to the Licence for details of permitted re-use. It is a condition of access that users recognise and abide by the legal requirements associated with these rights. If you believe that this work infringes copyright please provide details by email to qut.copyright@qut.edu.au

License: Creative Commons: Attribution-Noncommercial 4.0

Notice: *Please note that this document may not be the Version of Record (i.e. published version) of the work. Author manuscript versions (as Submitted for peer review or as Accepted for publication after peer review) can be identified by an absence of publisher branding and/or typeset appearance. If there is any doubt, please refer to the published source.*

https://doi.org/10.1145/3373017.3373023

Security Analysis of Fountain V1

MATTHEW BEIGHTON, Queensland University of Technology
HARRY BARTLETT, Queensland University of Technology
LEONIE SIMPSON, Queensland University of Technology

This paper analyses the security of the lightweight cryptographic algorithm Fountain (V1), which is a candidate in the current NIST competition for such ciphers. We examine the Boolean functions used in Fountain for state update and output. We show that correlations exist between S-box functions and some register stages that may lead to correlation attacks if certain update functions are detectable. We also show that the state update function avoids state convergence in any phase of cipher operation, but state collisions may be forced in one bit position, for select states, by manipulating the associated data or plaintext.

ACM Reference Format:
Matthew Beighton, Harry Bartlett, and Leonie Simpson. 2020. Security Analysis of Fountain V1. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW 2020), February 4–6, 2020, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373017.3373023

1 INTRODUCTION

Demands for smaller, faster and more complex technology within ever-shrinking, highly complex hardware, have focused attention on the design of lightweight cryptographic algorithms. In April of 2019, the United States National Institute of Security and Technology (NIST) launched a competition for new lightweight cryptographic algorithms, capable of authenticated encryption with associated data, that received 57 first round submissions [1].

One of these submissions was a binary feedback shift register based stream cipher named Fountain (v1), by Zhang [4]. Fountain uses a 128-bit secret key and a 96-bit initialisation vector as inputs to a keystream generator with a 256-bit state. Fountain provides both encryption and authentication for an input message. Encryption is performed by XOR-ing the plaintext message with a keystream. Authentication is provided by including a tag, generated by XORing the secret key with portions of keystream. Fountain provides authentication, but not encryption for the associated data.

To the best of the author's knowledge, there is currently only one analysis of Fountain in the public literature. Posteuca, in [2], analyses the existence of key-related slid pairs in Fountain and presents a slide attack. Posteuca claims the attack has a time complexity around 2^{80}. The practicality of this slide attack was disputed by Zhang, who stated that the proposed attack was based on the related key model. In the more general case where the key is unknown, the complexity of the attack becomes well above the claimed 2^{112} security level.

In this paper, we examine the security of Fountain against correlation attacks and demonstrate the existence of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
Manuscript submitted to ACM

state collisions in Fountain. This paper is organised as follows: Section 2 provides a description of Fountain. An overview of the phases of operation is given in Section 3. In Section 4 observations regarding the security of Fountain against correlation attacks are made. Section 5 investigates state convergence and state collisions in the different operating phases of Fountain. Finally, we conclude the paper in Section 6.

2 DESCRIPTION OF FOUNTAIN

This section describes the stream cipher Fountain, based on the specification submission to the NIST lightweight cryptography competition [4]. A diagrammatic representation of the internal state of Fountain, together with the state update function is shown in Figure 1.

Fountain's keystream generator consists of four regularly clocked 64-bit shift registers. There are several phases of operation: initialisation (which we denote as I), associated data (AD), diffusion (DI), keystream generation (KG) and finalisation/tag generation (FT). The update function for each register R is a linear combination of several independent functions: a linear function, taking inputs from four stages in R; a nonlinear function of four bits (one from each of the four registers) obtained through the application of an S-box; and, depending on the phase of operation, either or both of a nonlinear output function based on bits from all registers and an external input. This nonlinear output function is also used to provide keystream in the relevant phase of operation. Specific details of all functions are given below.

2.1 Notations

The following notation are used consistently throughout this paper:

Inputs and outputs:

- K denotes the 128-bit key, with key bits labeled $k_0, ..., k_{127}$, or represented in bytes as $K_0||K_1||\ldots||K_{15}$.
- V denotes the 96-bit initial vector, with bits labeled $v_0, ..., v_{95}$, represented in bytes as $V_0||V_1||\ldots||V_{11}$.
- Q denotes the 32-bit constant, with bits labeled $q_0, ..., q_{31}$, represented in bytes as $Q_0||Q_1||Q_2||Q_3$.
- $P = (p_0, ..., p_{n-1})$ is the plaintext message of length n, for $n \leq 2^{64}$.
- $C = (c_0, ..., c_{n-1})$ is the ciphertext message of length n, for $n \leq 2^{64}$.
- $H = (h_0, ..., h_{\ell-1})$ is the associated data, for $\ell \leq 2^{64}$.
- E denotes the 128-bit authentication tag, with tag bits labeled $e_0, e_1, ...e_{127}$.

Components:

- S^t is the overall state of Fountain at time t, where S consists of the contents of the four 64-bit shift registers, labeled A, B, G, D respectively.
- Register R consists of stages $r_0, r_1, \ldots, r_{63}$, where $R \in \{A, B, G, D\}$.
- The linear feedback functions for registers A, B, G, D are denoted f_A, f_B, f_G and f_D respectively.
- The 4x4 S-box transformation is considered as four Boolean functions. These form part of the functions used to update the shift registers. These functions are denoted $y_{A,j}, y_{B,j}, y_{G,j}$ and $y_{D,j}$ for $j \in \{I, AD, DI, KG, FT\}$. That is, the functions depend on the phase of operation.
- z^t is the nonlinear output function at time t.
- The combined feedback to each register $R \in \{A, B, G, D\}$ during phase j is given by $f_R \oplus y_{R,j} \oplus m$, where

$$m = \begin{cases} z & \text{for initialisation} \\ z \oplus h & \text{for associated data processing} \\ z & \text{for the diffusion phase} \\ p & \text{for plaintext processing} \\ z & \text{for finalisation and tag generation} \end{cases}$$

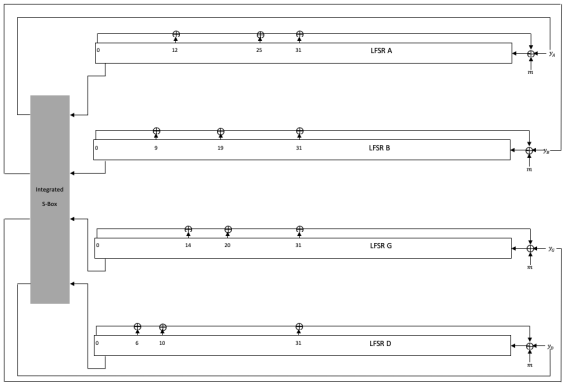

Fig. 1. The internal state of Fountain together with the update function.

2.2 Functions

As noted above, the internal state of Fountain is contained in four 64-bit binary shift registers. The overall state update function can be considered as four sub-functions, with each register having a different update function. In this section we detail the functions that make up the overall feedback for each register.

2.2.1 Linear update of registers. The shift registers have the following feedback polynomials and corresponding linear relations:

$$\begin{aligned} 0 &= 1 + x^{12} + x^{25} + x^{31} + x^{64} & f_A(x) &= a_0 \oplus a_{12} \oplus a_{25} \oplus a_{31} \\ 0 &= 1 + x^{9} + x^{19} + x^{31} + x^{64} & f_B(x) &= b_0 \oplus b_{9} \oplus b_{19} \oplus b_{31} \\ 0 &= 1 + x^{14} + x^{20} + x^{31} + x^{64} & f_G(x) &= g_0 \oplus g_{14} \oplus g_{20} \oplus g_{31} \\ 0 &= 1 + x^{6} + x^{10} + x^{31} + x^{64} & f_D(x) &= d_0 \oplus d_{6} \oplus d_{10} \oplus d_{31} \end{aligned}$$

These four feedback polynomials were chosen to be primitive. Furthermore, the linear functions were chosen such that the inputs are taken from the left-most 32 bits of each register. This choice was made to allow for parallel processing of 32 bits at a time. For autonomous LFSRs, these feedback functions would result in the production of sequences of guaranteed period length $2^{64} - 1$. As the registers are not autonomous, the benefit is not realised in Fountain.

2.2.2 Nonlinear functions. Fountain introduces nonlinearity into the state update in two ways: an S-box transformation and a nonlinear output function.

The integrated S-box. Fountain uses an integrated 4x4 S-box (for details see [4]) that takes four bits a_1, b_1, g_1, d_1 (from registers A, B, G and D respectively) as input and returns a 4-bit output. To enable comprehensive analysis we express the complete feedback function explicitly. As an intermediate step, we write the S-box as four, 4-1 Boolean functions, y_A, y_B, y_G, y_D, each with the same 4-bit input. Each of these output values is used in the update of one of the shift registers. As different S-boxes are provided for different phases of operation, we present the set of four Boolean functions for each phase in Tables 1 to 3 below.

Table 1. Boolean function representation of S-box transformation for initialisation and keystream generation.

$y_{A,I/KG}$	$1 \oplus g_1 \oplus d_1 \oplus a_1g_1 \oplus b_1d_1 \oplus a_1b_1d_1$
$y_{B,I/KG}$	$a_1 \oplus g_1 \oplus d_1 \oplus a_1g_1 \oplus a_1d_1 \oplus b_1d_1 \oplus g_1d_1 \oplus a_1b_1g_1$
$y_{G,I/KG}$	$a_1 \oplus g_1 \oplus a_1d_1 \oplus b_1d_1 \oplus g_1d_1 \oplus a_1b_1g_1 \oplus a_1b_1d_1$
$y_{D,I/KG}$	$1 \oplus a_1 \oplus b_1 \oplus g_1 \oplus a_1g_1 \oplus g_1d_1 \oplus a_1b_1d_1$

Table 2. Boolean function representation of S-box transformation for associated data and diffusion.

$y_{A,AD/DI}$	$1 \oplus a_1 \oplus a_1d_1 \oplus g_1d_1 \oplus a_1b_1g_1 \oplus a_1b_1d_1$
$y_{B,AD/DI}$	$a_1 \oplus g_1 \oplus d_1 \oplus a_1g_1 \oplus a_1d_1 \oplus b_1d_1 \oplus g_1d_1 \oplus a_1b_1g_1$
$y_{G,AD/DI}$	$a_1 \oplus g_1 \oplus a_1d_1 \oplus b_1d_1 \oplus g_1d_1 \oplus a_1b_1g_1 \oplus a_1b_1d_1$
$y_{D,AD/DI}$	$1 \oplus a_1 \oplus b_1 \oplus g_1 \oplus a_1g_1 \oplus g_1d_1 \oplus a_1b_1d_1$

Table 3. Boolean function representation of S-box transformation for finalisation.

$y_{A,FT}$	$1 \oplus b_1 \oplus a_1g_1 \oplus a_1d_1 \oplus b_1d_1 \oplus a_1b_1g_1$
$y_{B,FT}$	$b_1 \oplus g_1 \oplus d_1 \oplus a_1d_1 \oplus a_1b_1g_1 \oplus a_1b_1d_1$
$y_{G,FT}$	$1 \oplus a_1 \oplus a_1d_1 \oplus g_1d_1 \oplus a_1b_1g_1 \oplus a_1b_1d_1$
$y_{D,FT}$	$1 \oplus g_1 \oplus b_1d_1$

Output function. The output function z takes the contents of several stages from each of the four registers as inputs, as follows:

$$z = a_3 \oplus a_{11} \oplus b_{20} \oplus g_5 \oplus g_{16} \oplus d_7 \oplus d_{29} \oplus d_2a_5 \oplus b_4g_{11} \oplus d_{23}g_{27} \oplus b_{24}a_{29} \oplus d_2d_{23}d_{30}$$

2.2.3 Combined state update (generic). In every phase of operation, Fountain uses a state update function to transition from one time step to the next. State update is performed as follows:

For each register,
- Compute f_R, $y_{R,j}$ and z
- $r_i^{t+1} = \begin{cases} r_{i+1}^t & \text{for } i \in \{0, \ldots, 62\} \\ f_R \oplus y_{R,j} \oplus m & \text{for } i = 63, \end{cases}$

where m is phase dependent, as noted in Section 2.1

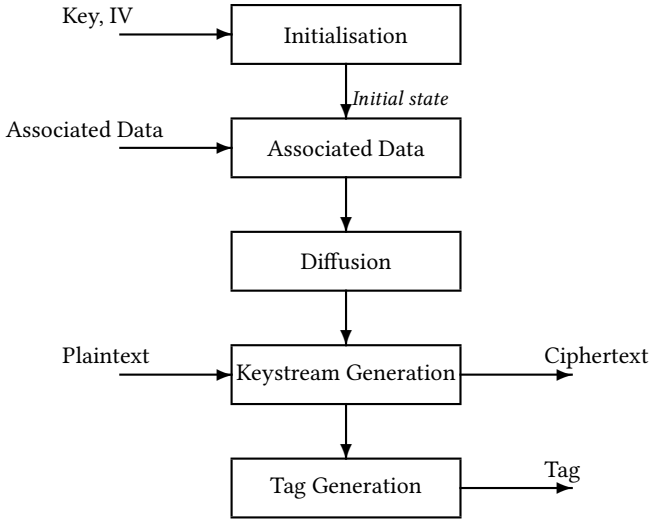

Fig. 2. Diagrammatic representation of the inputs and outputs throughout the operating phases of Fountain.

3 PHASES OF OPERATION

In this section, we describe the phases of operation in detail. A diagrammatic representation of the phases of operation, along with the inputs and outputs to each phase is provided in Figure 2.

3.1 Phase 1: Initialisation

Before keystream generation begins, the key and IV are loaded into the cipher and diffused throughout the state. In this section, we describe this process.

3.1.1 Loaded state. The key and IV are loaded into the first three registers (A, B, G) by alternating bytes of each, respectively. As the IV is shorter than the key, the final register is loaded with key bytes and the constant. After loading, the contents of each register are as follows:

Register A: $K_0||V_0||K_1||V_1||K_2||V_2||K_3||V_3$

Register B: $K_4||V_4||K_5||V_5||K_6||V_6||K_7||V_7$

Register G: $K_8||V_8||K_9||V_9||K_{10}||V_{10}||K_{11}||V_{11}$

Register D: $K_{12}||K_{13}||Q_0||K_{14}||K_{15}||Q_1||Q_2||Q_3$

We refer to this state as the *loaded state*. Note that the inclusion of constant values in register D imposes a format requirement on loaded states.

3.1.2 State update (initialisation). In the initialisation phase, state update is performed but no keystream is produced. Instead, the output bit is used as part of the feedback for each register. For initialisation, $m = z$ and $j = I$.

For $t = 0$ to 383, the state is updated using the generic state update function, given in Section 2.2.3. At the end of this process, the internal state will be referred to as the *initial state*.

3.2 Phase 2: Processing the associated Data

After the initialisation phase is completed, Fountain then processes the associated data H associated with the plaintext message. For associated data processing, $m = z \oplus h$ and $j = AD$.

For $t = 384$ to $383 + \ell$, the state is updated using the generic state update function.

3.3 Phase 3: Diffusion

Once the associated data has been processed, a diffusion process takes place. This involves 64 iterations of the state update function. Note that even if there is no associated data, these 64 steps still take place. For this process, $m = z$ and $j = DI$.

For $t = 384 + \ell$ to $447 + \ell$, the state is updated using the generic state update. Following this, b_0 is complemented to distinguish the change between processing the associated data and the plaintext.

3.4 Phase 4: Processing the plaintext

Once the diffusion phase is complete, Fountain is used to encrypt plaintext to form ciphertext. For this phase, $m = p$ and $j = KG$. Note that each plaintext bit is used in two different ways here: firstly, as an input to the state update function and secondly, combined with the keystream output bit to form the ciphertext. Thus, there are two actions taking place in this phase, state update and keystream generation.

For $t = 448 + \ell$ to $447 + \ell + n$, the state is updated using the generic state update. At the same time, the output function is applied to the internal state to produce keystream, which is XORed with the plaintext to produce ciphertext ($c = z \oplus p$). Following plaintext processing, d_1 is complemented to distinguish the change to finalisation/tag generation.

3.5 Phase 5: Finalisation and tag generation

The final phase of operation is tag generation. For this, $m = z$ and $j = FT$.

For $t = 448 + \ell + n$ to $831 + \ell + n$, the state is updated using the generic state update. The keystream output is discarded apart from the last 128 bits. The tag is generated by taking the XOR of the key, K, and the final 128 bits of keystream produced.

For $t = 0$ to 127, $E_i = k_i \oplus z^{704+\ell+n+i}$.

4 CORRELATIONS IN FOUNTAIN

In this section we explore correlations between the Boolean functions used in Fountain. We first represent the S-box transformation as four 4-1 Boolean functions. Although the S-boxes defined in the Fountain specification [4] are different for each phase, we observe that three of the four Boolean functions corresponding to the S-boxes remain the same across the initialisation, associated data, diffusion and plaintext processing phases. Our observations in this section come from the analysis of these Boolean functions.

4.1 Observations

A cryptographically strong Boolean function must exhibit a high degree of correlation immunity. In the following sections, we examine each $y_{R,j}$ for $R \in \{A, B, G, D\}$ over every phase of operation and make observations.

4.1.1 Phase 1: Initialisation. We observe the following about the S-box functions used in initialisation:

(1) None of these functions are correlation immune. In fact, during initialisation each $y_{R,I}$ for $R \in \{A, B, G, D\}$ is correlated to a single input. Specifically,
   $y_{A,I} = d_1$ with probability 0.25
   $y_{B,I} = b_1$ with probability 0.25
   $y_{G,I} = b_1$ with probability 0.75
   $y_{D,I} = g_1$ with probability 0.625.
(2) No function is correlated to a_1.
(3) The correlation between $y_{D,I}$ and g_1 is less biased than for the other functions.
(4) Examining the Walsh transform of $y_{D,I}$ we observe that $y_{D,I} = a_1 \oplus b_1$ with probability 0.25.
(5) Examining the Walsh transform of $y_{B,I}$ we observe that $y_{B,I} = a_1 \oplus g_1$ with probability 0.25.

4.1.2 Phase 2: Associated data. We observe the following about the S-box functions used when processing the associated data:

(1) None of these functions are correlation immune. In fact, during associated data processing each $y_{R,AD}$ for $R \in \{A, B, G, D\}$ is correlated to a single input. Specifically,
   $y_{A,AD} = a_1$ with probability 0.25
   $y_{B,AD} = b_1$ with probability 0.25
   $y_{G,AD} = b_1$ with probability 0.75
   $y_{D,AD} = g_1$ with probability 0.625.
(2) No function is correlated to d_1.
(3) The correlation between $y_{D,AD}$ and g_1 is less biased than for the other functions.
(4) Examining the Walsh transform of $y_{B,AD}$ we observe that $y_{B,AD} = a_1 \oplus g_1$ with probability 0.25.
(5) Examining the Walsh transform of $y_{A,AD}$ we observe that $y_{A,AD} = g_1 \oplus d_1$ with probability 0.75.

4.1.3 Phase 3: Diffusion. The diffusion phase uses the same Boolean functions as the associated data processing phase. Thus, all observations made about the functions used, in the diffusion phase, are the same as those noted in Section 4.1.2.

4.1.4 Phase 4: Keystream generation. When generating keystream, Boolean functions used are the same as the initialisation phase. Thus, all observations made about the functions when processing the keystream are the same as those noted in Section 4.1.1.

4.1.5 Phase 5: Finalisation and tag generation. We observe the following about the S-box functions used in finalisation:

(1) None of these functions are correlation immune. In fact, during finalisation, each $y_{R,FT}$ for $R \in \{A, B, G, D\}$ is correlated to a single input. Specifically,
   $y_{A,FT} = b_1$ with probability 0.25
   $y_{B,FT} = g_1$ with probability 0.625
   $y_{G,FT} = a_1$ with probability 0.25
   $y_{D,FT} = g_1$ with probability 0.25.
(2) No function is correlated to d_1.
(3) The correlation between $y_{B,FT}$ and g_1 is less biased than for the other functions, though $y_{D,FT}$ is correlated to g_1 with a higher bias.
(4) Examining the Walsh transform of $y_{D,FT}$ we observe that $y_{D,FT} = b_1 \oplus g_1 \oplus d_1$ with probability 0.75.

4.2 Security implications

The low correlation immunity of the Boolean functions used in the update of each register can be seen as a weakness and may be exploitable in a correlation attack. For such an attack to be successful, access to the values of $y_{R,j}$, for $R \in \{A, B, G, D\}$ and a particular $j \in \{I, AD, DI, KG, FT\}$ is required. This may be possible, for instance, through a side channel attack [3].

In the following sections we outline possible correlation attacks if the values of $y_{R,j}$ are known. The attack is performed in a divide and conquer manner, obtaining the values for one particular register at a time. We first provide an outline of an attack applied during initialisation and then an attack in the other phases. The obvious advantage of an attack on initialisation is that for a given IV, portions of registers A, B and G are known.

4.2.1 Initialisation. Assume that an attacker is able to detect the values of $y_{R,I}$ $R \in \{A, B, G, D\}$ for each time step in initialisation. Using the correlation between $y_{B,I}$ and b_1 along with the correlation between $y_{G,I}$ and b_1 , they could exhaustively search the 32 bits of key loaded into register B to find all possible candidates that satisfy these correlations. Similarly, using the correlation between $y_{A,I}$ and d_1, they could do the same for register D to find all possible candidates that satisfy this correlation. The attacker now has candidates for the loaded key in register B and in register D.

Now, using the correlation between $y_{D,I}$ and $a_1 \oplus b_1$, given that the attacker has the possible candidates for the key in register B, they could then exhaustively search the 32 bits of key loaded into register A to find all possible candidates that satisfy this correlation. Using the key candidates for register A, the attacker could use the correlation between $y_{B,I}$ and $a_1 \oplus g_1$ to exhaustively search the 32 bits of key loaded into register G to find all possible candidates that satisfy this correlation. The attacker would then have candidates for the entire key. These candidates could be tested with known plaintext to check which produces the correct keystream. The candidate that produces the correct keystream is the correct key.

In the case where the bits $y_{R,I}$ cannot be observed, the attack fails. Without a side channel, the observed output of Fountain is ciphertext. Firstly, in order to see the keystream, the attacker must know some plaintext. In order to recover any part of the key using the correlations in the S-box, the output function must be correlated to each $y_{R,I}$. But this is equivalent to the output function being correlated to the contents of stage 1 from each register. As the output function does not use the contents of stage 1 from any register directly, this is not possible. If initialisation has worked properly, the inputs to both the S-box and the output function will be uniformly distributed. Thus, the output function will not be correlated to $y_{R,I}$ and hence, not correlated to the contents of stage 1 from any register.

4.2.2 All other phases. If an attack reveals the values of $y_{R,j}$, for $R \in \{A, B, G, D\}$ and any phase $j \in \{AD, DI, KG, FT\}$, then a correlation attack can be performed analogous to the one described in Section 4.2.1. Unlike initialisation, in all other phases of operation the loaded state has been diffused. Therefore, instead of exhaustively searching 32 bits of key for each register, all 64 bits of each register would need to be exhaustively searched and checked against the correlations relevant to the phase of operation. Once candidates have been determined for all four registers in a particular phase, candidates for a particular state can be formulated.

We establish in Section 5.1 that the state update function is one-to-one. Thus, each candidate for the state can be used to clock the cipher back to time 0. Candidates that clock back to a state that is consistent with the format of a loaded state in Fountain as described in Section 3.1.1 (for the given IV), would reveal candidates for the correct key. These key candidates could be tested to identify the correct key.

As in initialisation, if the values of $y_{R,j}$, for $R \in \{A, B, G, D\}$ and for a particular phase $j \in \{AD, DI, KG, FT\}$ are not seen, the attack fails. Thus, the security of Fountain against correlation attacks ultimately depends on not being able to access the values of $y_{R,j}$, for $R \in \{A, B, G, D\}$, $j \in \{AD, DI, KG, FT\}$. That is, the implementation of Fountain must be resistant against side channel attacks.

5 STATE CONVERGENCE AND STATE COLLISIONS

State convergence occurs when two distinct states at time t map to the same state at time $t + 1$, under an autonomous update function. The phases of operation in Fountain that are autonomous are the initialisation, diffusion and finalisation phases. State convergence effectively reduces the number of possible internal states of the keystream generator over time.

In the case where the state update function is not autonomous, two distinct states at time t may map to the same state at time $t + 1$ for appropriate external input choices. We refer to this as a *state collision*. The phases of operation in Fountain that are not autonomous are the associated data and plaintext processing phases.

In the following two sections we investigate possibilities for state convergence and state collisions over these two sets of phases.

5.1 Convergence in phases with autonomous state update

During the initialisation, diffusion and finalisation phases, the state update function is given by

$$F_R = f_R \oplus y_{R,j} \oplus z, \text{ for } R \in \{A, B, G, D\}. \tag{1}$$

Convergence is possible if the overall feedback function is not one-to-one. In particular, given a state at time $t+1$, if the state at time t cannot be uniquely determined, state convergence is possible. We consider reversing the state update function. Moving from time $t+1$ back to time t, we have the following:

$$r_i^t = \begin{cases} u_R & \text{for } i = 0 \\ r_{i-1}^{t+1} & \text{for } 1 \leq i \leq 63 \end{cases}$$

where u_R, $R \in \{A, B, G, D\}$, is unknown. We now consider how to calculate the values of u_R.

Consider the state update function in the forward direction. From the equations given in Sections 2.2.1 and 2.2.2 we see that at each clock, the contents of r_0 for each register form part of the linear feedback f_R. Once used to generate the respective f_R, these bits are then shifted out of the register. **These are the only bits that are lost at each time step**. The contents of stage 63 for each register, at time $t+1$, is given by $F_R^t = f_R^t \oplus y_{R,j}^t \oplus z^t$, $R \in \{A, B, G, D\}$, the value of which is known. Note that although the functions $y_{R,j}$ used to generate F_R differ depending on the phase of operation, none of these functions depend on the contents of stage 0 and so the fact they differ is not relevant here. Clocking the registers backwards from time $t+1$ to t reveals the inputs to the functions $y_{R,j}^t$ and z^t. An equation for f_R^t can therefore be formulated as

$$f_R^t = F_R^t \oplus y_{R,j}^t \oplus z^t,$$

where, $F_R^t, y_{R,j}^t$ and z^t are known.
Moreover, the functions f_R^t are given by:

$$\begin{aligned} f_A^t &= a_0^t \oplus a_{12}^t \oplus a_{25}^t \oplus a_{31}^t \\ f_B^t &= b_0^t \oplus b_9^t \oplus b_{19}^t \oplus b_{31}^t \\ f_G^t &= g_0^t \oplus g_{14}^t \oplus g_{20}^t \oplus g_{31}^t \\ f_D^t &= d_0^t \oplus d_6^t \oplus d_{10}^t \oplus d_{31}^t \end{aligned}$$

We note that the inputs to each of these functions are known except for the contents of stage 0 for each register. Thus, an equation for each u_R, $R \in \{A, B, G, D\}$ can be generated.

$$\begin{aligned} u_A &= a_0^t = a_{12}^t \oplus a_{25}^t \oplus a_{31}^t \oplus F_A^t \oplus y_{A,j}^t \oplus z^t \\ u_B &= b_0^t = b_9^t \oplus b_{19}^t \oplus b_{31}^t \oplus F_B^t \oplus y_{B,j}^t \oplus z^t \\ u_G &= g_0^t = g_{14}^t \oplus g_{20}^t \oplus g_{31}^t \oplus F_G^t \oplus y_{G,j}^t \oplus z^t \\ u_D &= d_0^t = d_6^t \oplus d_{10}^t \oplus d_{31}^t \oplus F_D^t \oplus y_{D,j}^t \oplus z^t \end{aligned}$$

Solving these equations recovers the contents of stage 0 for each register. Therefore, S^t can be uniquely determined. We conclude that the state update function is invertible and therefore one-to-one. It follows that the state cycles of Fountain

are non-branching. Hence, there is no state convergence during the initialisation, diffusion or finalisation phases.

5.2 Collisions in phases with nonautonomous state update

Now consider associated data and plaintext processing phases. During these phases external inputs form part of the update to the internal state of Fountain. We show that state collisions can occur when processing the associated data. Applying the same logic, this is also clearly the case when processing the plaintext.

5.2.1 Collisions in associated data processing. When processing the associated data, the state update function is given by

$$F_R = f_R \oplus y_{R,AD} \oplus z \oplus h, \text{ for } R \in \{A, B, G, D\}, \tag{2}$$

where h is a bit of associated data corresponding to a particular time instant.

In this phase, the associated data is used in the overall feedback function. As the value of the associated data is known, it may be possible to force a state collision here. To see this, we must look at the inputs to the feedback functions.

Consider two states S^t and S'^t comprising registers $R \in \{A, B, G, D\}$ and $R' \in \{A', B', G', D'\}$ respectively. Suppose we have the following relationship between S^t and S'^t:

$$r_i'^t = \begin{cases} \overline{r_i^t} & \text{for } i = 0 \\ r_i^t & \text{for } i = 1, \ldots, 63, \end{cases} \tag{3}$$

for $r \in \{a, b, g, d\}$, where $\overline{x}$ denoted the binary complement of an element x.

As noted in Section 5.1, whenever the cipher is clocked, r_0 contributes to the linear feedback f_R and is then shifted out of the respective register. r_0 is not used as input to any other component of the update function. Because these stages are only used linearly in the overall feedback, complementing all four of these values will necessarily complement the overall feedback to each register. Thus, if we have that $S^t = S'^t$ for all stages except r_0, then at the next time step we can force a state collision by complementing the appropriate bit of associated data.

Table 4 provides an example of two Fountain states, S^t and S'^t, together with the corresponding external input bits h and h' that will collide at the next time instant. Note that S'^t can readily be formed from S^t by complementing the value in r_1 in each register and h' is simply the complement of h, as described in Equation 3
Now consider states S^t and S'^t that differ only in the contents of stage 1. That is,

$$r_i^t = \begin{cases} \overline{r_i'^t} & \text{for } i = 1 \\ r_i'^t & \text{for } i = 0, 2, 3, \ldots, 63, \end{cases}$$

for $r \in \{a, b, g, d\}$.

Forcing a state collision after two time steps will only be possible if the condition placed on the contents of stage 0 holds after one time step. That is, $r_1^t = \overline{r_1'^t}, r \in \{a, b, g, d\}$. Furthermore, the overall feedback bits at time t must be the

Table 4. Example of two distinct states that will collide at the next time instance by manipulating the associated data.

S^t	ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
h	1
F_A^t	0
F_B^t	0
F_G^t	1
F_D^t	1
S'^t	7f ff ff ff 7f ff ff ff 7f ff ff ff 7f ff ff ff
h'	0
F'^t_A	1
F'^t_B	1
F'^t_G	0
F'^t_D	0

same ($F_R^t = F_R'^t, R \in \{A, B, G, D\}$).

From Equation 2, along with the equations provided in Sections 2.2.1 and 2.2.2, we note that the only function with r_1 as an input is $y_{R,AD}$. Given that the contents of all stages are the same except for r_1, requiring $F_R^t = F_R'^t$ for $R \in \{A, B, G, D\}$ is equivalent to requiring:

$$y_{R,AD}^t \oplus h^t = y_{R,AD}'^t \oplus h'^t, \ \forall R \in \{A, B, G, D\}. \tag{4}$$

The value of h^t and h'^t can be manipulated, but will be the same for all R in each state . Therefore, in order to be able to force Equation 4 to hold for all R, we have that $y_{R,AD}^t = y_{R,AD}'^t$ or $y_{R,AD}^t = \overline{y_{R,AD}'^t}$. Given that the inputs to $y_{R,AD}^t$ and $y_{R,AD}'^t$ are restricted to complementary pairs, Equation 3 can be satisfied if $\exists x \in \{0,1\}^4$ such that $y_{R,AD}(\overline{x}) = \overline{y_{R,AD}(x)}$ $\forall R \in \{A, B, G, D\}$.

Table 5 presents the truth table of each nonlinear function $y_{R,AD}, R \in \{A, B, G, D\}$, reformatted in a way that presents complementary inputs on the same row. The output of each function has been concatenated so that each 4-bit input has a corresponding 4-bit output. For example, looking at the first row of Table 5, we see that for the input vector 0000, the outputs are:

$y_{A,AD} = 1$, $y_{B,AD} = 0$, $y_{G,AD} = 0$, $y_{D,AD} = 1$.

Looking at the output columns in each row, we see that complementary inputs never produce complementary outputs. Since $\nexists x \in \{0,1\}^4$ such that $y_{R,AD}(\overline{x}) = \overline{y_{R,AD}(x)}$ $\forall R \in \{A, B, G, D\}$, it is not possible to force state collisions for states that differ in the contents of stages 1 and therefore, for states that differ in the contents of stage 0 and 1. Thus, forcing a state collision is only possible when two states differ in the contents of stage 0, in each register. For each of the 2^{256} states, there is exactly one other state for which this can occur.

5.2.2 Collisions in plaintext processing. The same logic can be applied to the plaintext processing phase, as the plaintext bits form the external input to the registers feedback functions. As for associated data processing, the way r_0 is used in

Table 5. Reformatted truth table for $y_{R,AD}$, $R \in \{A, B, G, D\}$ to compare pairs of complementary inputs and their outputs.

input 1	output 1	input 2	output 2
0000	1 0 0 1	1111	0 0 1 1
0001	1 1 0 1	1110	1 0 1 1
0010	1 1 1 0	1101	0 0 0 0
0011	0 1 0 1	1100	0 1 1 1
0100	1 0 0 0	1011	0 0 0 1
0101	1 0 1 0	1010	0 1 0 0
0110	1 1 1 1	1001	1 1 0 0
0111	0 0 1 0	1000	0 1 1 0

the feedback register R means that a state collision can be forced for two states which differ in $r_0, r \in \{a, b, g, d\}$. When processing the plaintext, the output function z is not used in the overall feedback function of any register. This does not impact the process described in Section 5.2.1 as z does not take the r_0 or r_1 input.

In Table 6 we have reformatted the truth table of each nonlinear function $y_{R,KG}, R \in \{A, B, G, D\}$, analogous to the reformatting done for associated data processing. We observe that the S-box used in this phase also does not satisfy the necessary condition of sending complementary inputs to complementary outputs. Therefore, when processing the plaintext, a state collision is only possible when two states differ in the contents of stage 0, in each register, by negation. For each of the 2^{256} states, there is exactly one other state for which will this occur.

Table 6. Reformatted truth table for $y_{R,KG}$, $R \in \{A, B, G, D\}$ to compare pairs of complementary inputs and their outputs.

input 1	output 1	input 2	output 2
0000	1 0 0 1	1111	0 0 1 1
0001	0 1 0 1	1110	1 0 1 1
0010	0 1 1 0	1101	0 0 0 0
0011	1 1 0 1	1100	1 1 1 1
0100	1 0 0 0	1011	0 0 0 1
0101	1 0 1 0	1010	1 1 0 0
0110	0 1 1 1	1001	0 1 0 0
0111	0 0 1 0	1000	1 1 1 0

5.2.3 Security implication and discussion. In Sections 5.2.1 and 5.2.2, we showed that if two distinct states differ only in the contents of stage 0, in each register, a single bit of associated data (or plaintext) can be manipulated to force a collision. Collisions forced in this way cannot be avoided without allowing state convergence, since it relies on the contents of stage 0 from each register being used linearly in the overall feedback function.

It is known from Section 5.1 that the state update function is invertible. Thus, if the plaintext and/or the associated data is known (depending on the phase), given any state at time t, a user can clock the cipher backwards to identify the corresponding loaded state. Consider the possibility of sending two distinct messages that produce the same tag. For convenience, consider an initial state S^{384}. If we wish to change the associated data bit at time 384 and still produce the same tag, we must find a state S'^{384} that satisfies Equation 3. To do this, we simply complement the contents of stage 0

in each register of S^{384} to produce S'^{384}. We now have a state S'^{384} such that complementing the original associated data bit h at time 384, will produce the same tag as S^{384}.

The state S^{384} was generated by taking a loaded state S^0 and clocking the cipher 384 times. Every loaded state in Fountain follows a particular format; the 32-bit constant must be in the same position every time. Thus, if S'^{384} is a valid state, it must clock back to a valid loaded state. That is, if S'^0 respects the format of loaded states in Fountain, we will have a key, IV pair (K, V), such that complementing the associated data bit at time $t = 384$ will produce the same tag for S^0 and S'^0. The probability of the state S'^0 will be a valid loaded state is 2^{-32} (assuming initialisation has worked correctly). Thus, is is very possible that the state S'^{384} may not clock back to a valid loaded state. Furthermore, the nonlinearity introduced by the S-box and the output function, together with the distribution of the linear update taps, will make the relationship between the two loaded states extremely complex. It is therefore likely that even if S'^0 is valid, the key and IV used to load this state will differ from that used in S^0. In this case, though a forgery is possible, the change in K and V would raise suspicion in the receiver.

The restrictions imposed by the use of the contents of stage 0 and the S-box transformations in the design of Fountain means that forcing state collisions at more than one time step is not possible. This, together with the fact that identifying states that differ in the contents of stage 0 is difficult makes forgeries hard in practice.

6 CONCLUSION

In this paper, we analysed the security of the stream cipher Fountain by examining the Boolean functions used for state update and output. We considered the possibility of correlation attacks, state convergence and state collisions in the different phases of operation. We showed that if the output of the S-box transformation is detectable (for example, via a side channel attack), a correlation attack is possible with full key recovery. The security of Fountain against correlation attacks therefore relies on its ability to resist side channel attacks.

We also showed that state convergence is not possible in phases of operation with autonomous state update (initialisation, diffusion and finalisation) due to the invertibility of the overall state update function. In the phases where the associated data and plaintext are processed, we found that forcing state collisions are theoretically possible, but only for a single time step. In practice, the states for which a collision is possible can be identified, but it is not clear that both of these states come from valid loaded states. Further, it is not obvious how to force two arbitrary states to collide after any number of clocks.

REFERENCES

[1] NIST. 2019. Lightweight Cryptography. *NIST Information Technology Laboratory, CSRC, Lightweight Cryptography, Round 1 Candidates* (2019). https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-1-Candidates

[2] Raluca Posteuca. 2019. Related-Key Differential Slide Attack Against Fountain V1. *Cryptology ePrint Archive, Report 2019/920* (2019). https://eprint.iacr.org/2019/920

[3] Christian Rechberger and Elisabeth Oswald. 2004. Stream ciphers and side-channel analysis. In *In ECRYPT Workshop, SASC-The State of the Art of Stream Ciphers*. Citeseer, 320–326.

[4] Bin Zhang. 2019. Fountain: A Lightweight Authenticated Cipher (v1). *NIST Lightweight Cryptography Competition* 1 (2019).