Attacks and Remedies for Randomness in AI (and with AI)

Cryptanalysis of PHILOX and THREEFRY

Jens Alich[1], Thomas Eisenbarth[2], Hosein Hadipour[1], Stefan Kölbl[3], Gregor Leander[1], Felix Mächtle[2], Yevhen Perehuda[1], Shahram Rasoolzadeh[1], Jonas Sander[2] and Cihangir Tezcan[4]

[1] Ruhr University Bochum, Bochum, Germany
[2] University of Lübeck, Lübeck, Germany
[3] Google, Switzerland
[4] Middle East Technical University, Ankara, Turkey

Abstract

In this work, we address the critical yet understudied question of the security of the most widely deployed pseudorandom number generators (PRNGs) in AI applications. We show that these generators are vulnerable to practical and low-cost attacks by introducing both manual and automated cryptanalysis techniques. For this we partially, and to the best of our knowledge for the first time, take advantage of AlphaEvolve, a novel evolutionary coding agent guided by Large Language Models, to automate the discovery of complex cryptographic distinguishers. Finally, we present a cryptographically secure and well-understood alternative, which has a negligible effect on the overall AI/ML workloads. More generally, we recommend the use of cryptographically strong PRNGs in all contexts where randomness is required, as past experience has repeatedly shown that security requirements may arise unexpectedly even in applications that appear uncritical at first.

Keywords: PRNG, GPU, Philox, Threefry, Speck, differential-linear cryptanalysis, multiplicative differentials, AlphaEvolve

1 Introduction

Machine learning systems are now prevalent in products and scientific workflows [HJA20, MSL+25, VSP+17, WAW+25, KFP+25] with many of their core algorithms heavily depending on randomness during training and inference. Sampling-based decoding in large language models (LLMs) [KFP+25], stochastic seeding in image [HJA20] and video [WAW+25] generators, and noise for differentially private (DP) training [DR14] are all driven by pseudorandom number generators (RNGs). As the outputs of these systems (e.g. trained models or generated media) and the cryptographic claims attached to them (e.g. authorship watermarks) are increasingly targeted by attacks, the demand for robust and cryptographically sound RNGs has surged [DSA24, MSS+26, GKN+26].

Emails: jens.alich@rub.de (Jens Alich), thomas.eisenbarth@uni-luebeck.de (Thomas Eisenbarth), hossein.hadipour@rub.de (Hosein Hadipour), kste@mailbox.org (Stefan Kölbl), gregor.leander@rub.de (Gregor Leander), f.maechtle@uni-luebeck.de (Felix Mächtle), yevhen.perehuda@rub.de (Yevhen Perehuda), shahram.rasoolzadeh@rub.de (Shahram Rasoolzadeh), j.sander@uni-luebeck.de (Jonas Sander), cihangir@metu.edu.tr (Cihangir Tezcan)

Date of this document: 2026-09-08.

Table 1: PRNG algorithms used by major machine learning frameworks and libraries across hardware platforms. Abbreviations: P = Philox-4×32 (10 rounds), P7 = Philox-4×32 (7 rounds), M = MT19937, T = Threefry-2×32 (20 rounds), A = AES-128-CTR, L = 64-bit Linear-Feedback Shift Register, X = xoroshiro128aox. A dash (–) denotes no support. Unless noted, seed size is 64 bits. Experimental versions are omitted.

Framework	GitHub Stars	Nvidia	x86	Apple M-Series	Nvidia Jetson	AWS Trn/Inf	Intel Gaudi	Graphcore IPU
TensorFlow [Goo25a]	194k	P/T	P/T	P/T	P/T	P[b]/T[b]	–	P/T
PyTorch [PyT25]	97.2k	P	M[b]	P/M[b]	P/M[b]	P[b]/T[b]	P/L	P/M[b]/X
JAX [Goo25b]	34.8k	T	T	T	T	–	–	T
PaddlePaddle [Bai25b]	23.6k	P	M	M	P	–	–	–
FlashAttention [DFE+25]	22.1k	P7	–	–	–	–	–	–
MXNet [The23]	20.8k	P[b]	M[b]	–	P[b]	P[b]/T[b]	–	–
NVIDIA Apex [NVI25a]	8.9k	P7	–	–	–	–	–	–
Opacus [Met25]	1.9k	P/A[a]	M[b]/A[a]	M[b]/P/A[a]	M[b]/P/A[a]	–	–	–
AMD ROCm [LZQ+]	516	–	P7	–	–	–	–	–

[a] Secure but without seed or a 32-bit seed.
[b] The key size is 32 bits.

Table 1 summarizes the RNGs exposed by widely used machine learning frameworks, i.e., TensorFlow [AAB+16], PyTorch [PGM+19], JAX [BFH+18], PaddlePaddle [Bai25a], FlashAttention [DFE+25], MXNet [CLL+15], NVIDIA Apex [NVI25a], Opacus [YSS+21] and AMD ROCm [LZQ+] across common hardware targets. Counter-based generators designed for parallelism, i.e., Philox and Threefry, dominate accelerator backends, reflecting their suitability for vectorized kernels and reproducible, splittable streams. In contrast, CPU architectures still use MT19937, Xoroshiro variants, and even LFSRs. These algorithms are attractive for their speed, but offer weak or no cryptographic protection. AES-128 appears only in the DP library Opacus and comes with seeding constraints: Either an unseeded secure mode or a 32-bit seed is used. However, disabling seeding is untenable for modern ML workflows: It undermines reproducibility and prevents practitioners from fixing seeds to obtain stable outputs, e.g., when iterating on prompts for image generation.

The same ML script can use different RNGs on different backends, with different reproducibility and security properties hidden behind the same API.

In this paper, we argue that contemporary ML stacks need an RNG primitive that is secure by construction and performant under realistic loads.

Unexpected Security Demands. Security expectations around randomness in ML workloads have evolved in ways that early designers of RNGs did not anticipate. Attacks such as prompt stealing in generative models [MSS+26] or PRNG-based exploits in randomized smoothing [DSA24] exploit the fact that existing generators are predictable or miss-seeded – issues that were initially considered irrelevant in non-adversarial settings like simulation or graphics. Moreover, recent defenses such as NoisePrints [GKN+26], a seed-based authorship watermark, explicitly rely on a cryptographically strong PRNG. A distinguishable or predictable generator would directly enable watermark forgery and removal. In practice, the randomness infrastructure of ML systems has quietly become part of their security perimeter.

This gap is also reflected in the design philosophy of widely adopted counter-based RNGs. As highlighted in [SMDS11], the newly designed PRNGs, Philox and Threefry were engineered for parallelism, statistical quality, and reproducibility. Their evaluation criterion was *crush-resistance*, that is the ability to pass the stringent BigCrush statistical tests from TestU01 in [LS07], rather than resistance to adversarial prediction or state compromise. The authors explicitly note that these generators are not cryptographically secure, but merely *good enough* as long as they withstand statistical batteries. What was once a reasonable compromise for HPC or Monte Carlo workloads now falls short when the same generators underpin sensitive ML tasks with adversarial threat models, such as

prompt stealing.

Thus, ML systems increasingly require cryptographically secure randomness, while their dominant PRNGs were selected mainly for speed, parallelism, and statistical quality.

Our Contribution.

In our work, we address the critical yet understudied question of the security of the most widely deployed PRNGs. As we show, these not only fail to deliver the promised security, but are also vulnerable to practical, low-cost attacks. To remedy the situation we finally present a cryptographically secure and well-understood alternative.

Efficiency Requirements. To better understand the efficiency requirements imposed by modern ML applications, we investigated how much randomness is required and how the demand is distributed over the application runtime. We run a study including four representative ML workloads, including LLM-based text generation, diffusion-based image generation, text-to-video generation, and privacy-preserving neural network training. As detailed in Section 2, we found that depending on the application, the randomness demand is either required once at the beginning or steadily spread over time. Therefore, a suitable RNG should enable low-latency sampling to accommodate both short, high-volume bursts and steady per-step demands. Additionally, we found the overhead induced by currently applied RNGs to be negligible compared to the overall runtime of these ML workloads, leaving a room to strengthen the robustness of the used RNGs without significantly affecting the end-to-end application runtime.

There are GPU-optimized table-based implementations of AES [Tez21], arguably the most conservative choice of a primitive, available. While these provide impressive and competitive runtime performance when a large amount of randomness is required at once, they offer less flexibility, as they occupy almost the entire shared memory of the GPU and thus make context switches rather inefficient. Therefore, in line with the basic design ideas of Threefry and Philox, we instead investigate possible alternatives based on operations that are especially efficient on modern GPUs.

Security of Current Standards. We focus on the main and most relevant cases, Philox and Threefry, proposed in [SMDS11]. Threefry is based on a known cryptographic primitive, Threefish, the block cipher used in the Skein hash function, one of five finalists in the NIST hash function competition. In fact, Threefry is essentially a round-reduced and shortened version of Threefish. Philox, on the other hand, is a more non-standard design that mainly uses the multiplication of 32-bit words in a Feistel-like structure as its round function.

The main security problem with widely used variants of both constructions is the far too short key length of 64 bits. This is especially problematic since, being optimized for GPUs (for good reasons), brute-forcing the used key is particularly cheap on exactly those devices. Following the recent work [TL25], we show that their 64-bit-key variants can be brute-forced at practical cost on commodity GPUs. The full details of our GPU attacks are in Appendix A.

However, the short key length is not the only security shortcoming we found. Both ciphers are vulnerable to attacks with practical time complexity, thereby violating the security properties claimed by their designers. Technically, the attacks are very different in nature and interesting in their own right.

Low Cost Attacks on Philox: Automated Cryptanalysis via LLM-Guided Evolutionary Search. For Philox, we develop practical key-recovery attacks using differential-linear and multiplicative-differential techniques, including a *ciphertext-only* attack that applies

in a realistic counter-based PRNG setting. We present practical attacks on 7-round Philox−4×32 requiring data complexity of only outputs for about 2,500 consecutive inputs, and time complexity of about $2^{34.23}$ encryptions.

Moreover, we demonstrate that LLM-guided evolutionary search frameworks, such as AlphaEvolve [NVE+25], can discover complex, non-intuitive cryptographic distinguishers. By adapting AlphaEvolve to the differential-linear cryptanalysis of Philox−4, we automatically discovered an 8-round distinguisher with a high bias of 0.66, significantly improving upon our manual cryptanalysis. This highlights a novel, highly efficient methodology for evaluating such designs and, to our knowledge, represents the first application of this approach to cryptanalysis.

By applying this 8-round (multiplicative) differential-linear distinguisher, we present a full key-recovery attack on 10-round Philox−4×32. This attack requires $2^{43.29}$ chosen plaintext-ciphertext pairs and a total time complexity of $2^{43.46}$ encryptions, which is primarily dominated by the data generation phase.

Differential-Linear Distinguishers for Threefry via a Generic Top-down Model. We present differential-linear distinguishers for round-reduced Threefry obtained through a *segmented*, top-down search methodology in Section 5. Beyond the resulting distinguishers themselves, our main contribution is the *model* that produces them: instead of building a bottom-up MILP/CP formulation of the cipher from individual components (addition, XOR, rotations, etc.) [HDE24, BGG+23], we treat the primitive as a black-box and infer strong DL correlations directly from its input-output behavior.

The approach is simple to implement (it only requires round-reduced encryption/decryption calls), portable across designs with similar round/key schedules, and can be applied beyond Threefry. Using this framework, we identify high-bias differential-linear distinguishers for Threefry reduced to 14 and 15 rounds (out of 20), with correlations $2^{-16.4}$ and $2^{-24.0}$, respectively.

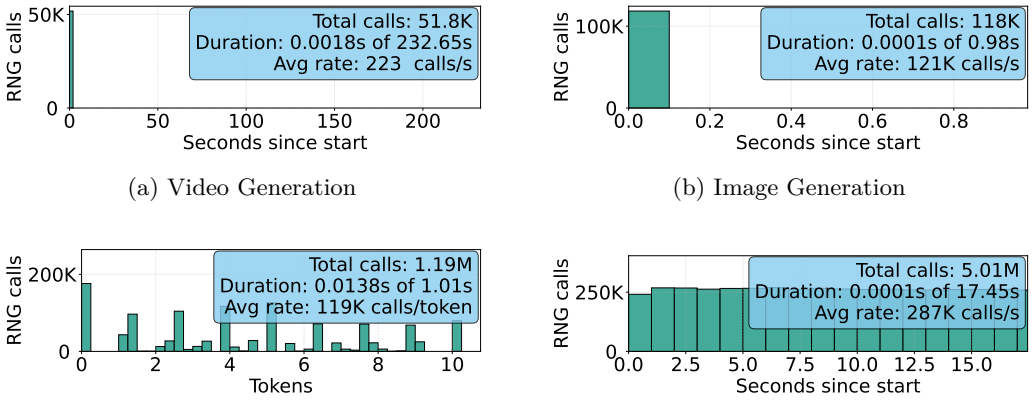
These insights on both Threefry and Philox show that merely increasing the key size by applying a suitable key-scheduling algorithm does not result in cryptographically secure designs. Given that the current methods fail to provide the security required in practice, there is a clear need for a secure and efficient alternative, ideally based on well-scrutinized primitives.

We therefore propose Polster, a GPU-friendly counter-based PRNG built from a conservative Speck-64/128 variant.

Polster: A Suitable and Conservative Proposal. Finally, motivated by the security limitations of Threefry and Philox, as well as our analysis of the actual requirements for GPU-based PRNGs, we propose a modified, round-reduced variant of a Speck block cipher [BSS+13]. Our design employs a 64-bit block size and, crucially, a 128-bit key, making it a practical and secure alternative to the existing constructions. In Section 6, we discuss both the implementation and security aspects of this choice, and introduce Polster, which is essentially Speck-64-128 reduced to 20 rounds with an improved key schedule, while still maintaining a sufficient security margin.

2 Requirements for Randomness in ML

To identify the requirements for a secure RNG used in ML, we start by examining how randomness is consumed across four representative ML workloads: text generation with a large language model (LLM), image generation with a diffusion model, text-to-video generation, and privacy-preserving training with differential privacy. These use cases cover the dominant ways randomness enters large-scale ML systems: sampling from discrete



(a) Video Generation

(b) Image Generation

(c) Text Generation

(d) Differentially Private Training

Figure 1: Temporal distribution of PRNG draws across four ML workloads. Measurements use PHILOX (except for text generation that uses MT19937 per framework default). Results are averaged over 100 runs on an NVIDIA A100 GPU.

distributions during LLM decoding, seeding a diffusion model, and injecting calibrated noise during training to achieve formal privacy guarantees.

During auto-regressive decoding, an LLM assigns a probability p_i to each possible next token index i, and one token is picked at random according to these probabilities. Two sampling methods are commonly used for the selection. *Inverse-transform sampling* draws a single random value $u \sim \text{Uniform}[0,1]$ and outputs the first token index i satisfying $\sum_{j<i} p_j < u \leq \sum_{j \leq i} p_j$. The *Gumbel-Max trick* draws independent Gumbel$(0,1)$ noises g_i and selects the next token index via $\arg\max_i\{\log p_i + g_i\}$, requiring one random draw per candidate token [JGP17]. In the following evaluation, we adhere to the popular HuggingFace Transformer Library [Fac25], which leverages the latter, meaning the per-token randomness consumption scales with the vocabulary size.

In diffusion-based image and video generation, a high-dimensional noise vector is sampled to seed the generation process [HJA20, WAW+25]. The subsequent denoising steps are essentially deterministic. In differentially private model training, gradients are clipped and then perturbed with additive noise (typically Gaussian) at every optimization step [DR14].

To quantify these randomness usage patterns of all four techniques, we instrumented each workload with a custom RNG hook that counts and timestamps PRNG calls. Concretely, we generated ten tokens with Google's Gemma-3 [KFP+25], produced images with Stable Diffusion 3.5 Turbo [Sta24], created videos with Wan 2.1 [WAW+25], and trained a digit classifier on MNIST [LeC98] using Opacus [YSS+21], a DP library based on PyTorch [PGM+19]. The cumulative count of draws and their timing are reported in Figure 1.

The resulting profiles are consistent with the description above. Diffusion and video models exhibit a sharp spike at the beginning when the initial noise is drawn and afterwards no further randomness is required. In total, the share of the end-to-end runtime attributable to RNG is a small fraction of the total generation time. LLM decoding consumes randomness on every token in the sequence, producing one spike per token as the tokens are sampled. Differentially private training requires fresh randomness on every batch, yielding a fairly constant draw rate over the course of training.

Recent work emphasized the urgent need for cryptographically secure RNGs in ML: Without secure randomness generation, Mächtle *et al.* [MSS+26] show that stealing the initial random noise used for image generation can directly result in stealing private user

prompts. Once the key is recovered – either by brute force or faster using our other cryptanalytic results - an adversary can regenerate randomness and reveal the prompt. Notably, prompt stealing were classified as (highly) confidential with important severity by Microsoft's Vulnerability Severity Classification for AI Systems [Mic23].

Similarly, Dahiya *et al.* [DSA24] show that randomness must be secure for ML safety and security guarantees, such as robustness certification, to hold.

Beyond confidentiality and robustness, randomness also underpins emerging integrity guarantees in generative ML: Goren *et al.* [GKN+26] propose NoisePrints, a watermarking scheme in which the seed of the initial diffusion noise itself acts as the proof of authorship. Here, both the unforgeability of the watermark and the secrecy of the seed reduce directly to the cryptographic strength of the underlying PRNG. A distinguishable or predictable generator opens the door to forgery and watermark stripping.

Beyond cryptographic strength, our observations above motivate the following additional requirements: The RNG should provide low-latency sampling to accommodate both short, high-volume bursts (e.g., at initialization of image/video generation) and the steady per-step demands of DP and LLM decoding. It should support many independent, reproducible streams across devices, and it should offer deterministic seeding to enable exact replay. Given that random number generation accounts for only a small share of wall-clock time in these workloads, a modest performance overhead is acceptable in exchange for stronger security and reproducibility.

3 Preliminaries

3.1 Notations

Let $\mathbb{F}_2$ denote the finite field with two elements $\{0,1\}$. For $n \in \mathbb{N}$, we write $\mathbb{F}_2^n$ for the n-dimensional vector space over $\mathbb{F}_2$. We identify $\mathbb{F}_2^n$ with the set of n-bit strings $\{0,1\}^n$.

Throughout the paper, we adopt the convention that the *least significant bit* (LSB) of a binary string is its rightmost entry. Formally, let $X = (x_{n-1}, \ldots, x_1, x_0)$ denote an n-bit vector, where the indexing begins at 0. Then the LSB is x_0, which we denote by $\mathrm{LSB}(X) = x_0$.

Bitwise XOR corresponds to addition in $\mathbb{F}_2^n$ and is denoted by $\oplus$. Hence, for $X, Y \in \mathbb{F}_2^n$, the $X \oplus Y$ is the vector whose i-th component is $x_i + y_i$ modulo 2.

When we write $a + b$ (resp. $a \times b$), the operation is modular addition (resp. modular multiplication) in $\mathbb{Z}_{2^n}$, i.e.,

$$a + b \equiv a + b \pmod{2^n}, \qquad a \times b \equiv a \cdot b \pmod{2^n}.$$

A block cipher with block size n is a family of keyed permutations $\{E_k\}_{k \in \mathcal{K}}$ with $E_k : \mathbb{F}_2^n \to \mathbb{F}_2^n$, where k ranges over the key space $\mathcal{K}$.

3.2 Philox Family of Block Ciphers

Philox is a family of block ciphers that follows a Feistel-like structure, but with an unconventional design choice: the use of integer multiplication within the round function. Each instance of the cipher is denoted by Philox–N×W–R, where N is the number of words (or branches) in the state, W is the bit size of each word and R is the number of rounds. The recommended parameters for N and W are $(\mathrm{N}, \mathrm{W}) \in \{(2, 64), (4, 64), (4, 32)\}$.

The block size of each variant is N × W bits, while the key size is only (N/2) × W bits, i.e., half the block size. Figure 2 illustrates the round functions of Philox–2×W (left) and Philox–4×W (right).

The core operation of the round function is the multiplication of one state word by a fixed constant. Let $M \in \mathbb{F}_2^{\mathrm{W}}$ denote this constant. Then, for a state word $X \in \mathbb{F}_2^{\mathrm{W}}$, the

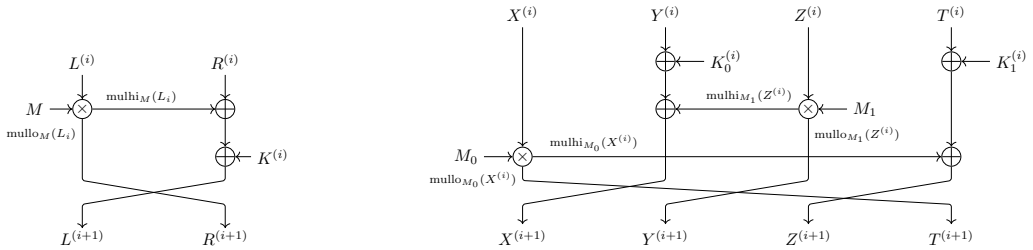


Figure 2: Philox–2×W (left) and Philox–4×W (right) round functions.

multiplication is defined as

$$\mathrm{mulhi}_M(X) := \lfloor (X \times M)/2^{\mathrm{W}} \rfloor\,, \quad \mathrm{mullo}_M(X) := (X \times M) \bmod 2^{\mathrm{W}},$$

where the high- and low-halves of the multiplication are subsequently used in the round functions.

It is well known that for an odd constant M, the function $\mathrm{mullo}_M(\cdot)$ is a bijection over $\mathbb{F}_2^{\mathrm{W}}$. We denote its inverse by $\mathrm{mullo}_M^{-1}(\cdot)$.

The round function of Philox–2×W operates on a 2W-bit state $(L^{(i)}, R^{(i)}) \in \mathbb{F}_2^{\mathrm{W}\times 2}$ together with a W-bit subkey $K^{(i)}$ as follows:

$$L^{(i+1)} = \mathrm{mullo}_M(R^{(i)})\,, \quad R^{(i+1)} = \mathrm{mulhi}_M(R^{(i)}) \oplus L^{(i)} \oplus K^{(i)}\,.$$

All round functions use a fixed odd multiplier; for Philox–2×64, this constant is $M =$ `0xD2B74407B1CE6E93`.

The round function of Philox–4×W operates on a 4W-bit state $(X^{(i)}, Y^{(i)}, Z^{(i)}, T^{(i)}) \in \mathbb{F}_2^{\mathrm{W}\times 4}$ together with two W-bit subkeys, $K_0^{(i)}$ and $K_1^{(i)}$, to compute the next state $(X^{(i+1)}, Y^{(i+1)}, Z^{(i+1)}, T^{(i+1)})$ as follows:

$$X^{(i+1)} = \mathrm{mulhi}_{M_1}(Z^{(i)}) \oplus Y^{(i)} \oplus K_0^{(i)}\,, \qquad Y^{(i+1)} = \mathrm{mullo}_{M_1}(Z^{(i)})\,,$$
$$Z^{(i+1)} = \mathrm{mulhi}_{M_0}(X^{(i)}) \oplus T^{(i)} \oplus K_1^{(i)}\,, \qquad T^{(i+1)} = \mathrm{mullo}_{M_0}(X^{(i)})\,.$$

Philox–4×W uses two different fixed odd multipliers in each round function. For Philox–4×32, these constants are $M_0 =$ `0xD2511F53` and $M_1 =$ `0xCD9E8D57`; and for Philox–4×64, they are

$$M_0 = \texttt{0xD2E7470EE14C6C93} \quad \text{and} \quad M_1 = \texttt{0xCA5A826395121157}\,.$$

The key schedule of Philox is deliberately simple and well suited for efficient implementation on GPU platforms. The N/2 subkeys are updated in each round according to

$$K_j^{(i+1)} = K_j^{(i)} + C_j\,, \quad \text{with } K_j^{(0)} = K_j\,,$$

for each $j \in \{1, \ldots, \mathrm{N}/2\}$, where the K_j are the N/2 words of the user key, and the C_j are fixed constants derived from $(\sqrt{3}-1)$ and the golden ratio. In particular, the designers specify $C_0 =$ `0x9E3779B9` and $C_1 =$ `0xBB67AE85` for W = 32, and $C_0 =$ `0x9E3779B97F4A7C15` and $C_1 =$ `0xBB67AE8584CAA73B` for W = 64.

In the original paper [SMDS11], the authors report that, with the recommended multipliers, Philox–2×64–6 achieves *full crush-resistance*, and they propose Philox–2×64–10 as an alternative with an additional safety margin. Besides, for Philox–4×W, the designers found that both word sizes, the cipher achieve full crush-resistance with 7 rounds.

Nowadays, the absolute majority of ML libraries adopt a 7- or 10-round version of Philox–4×32. Although, for example, NVIDIA's basic library cuRANDDx documentation explicitly recommends setting the round count for Philox to 7 (instead of the default 10) for better performance [NVI25b].

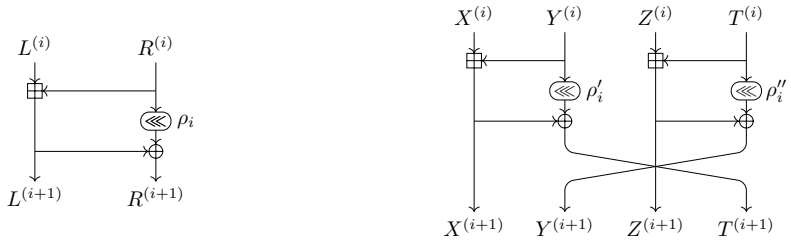


Figure 3: Threefry–2×W (left) and Threefry–4×W (right) round functions.

Table 2: The rotation constants relevant to the Threefry variants.

Variant	0	1	2	3	4	5	6	7
2x32: ρ_i	13	15	26	6	17	29	16	24
2x64: ρ_i	16	42	12	31	16	32	24	21
4x32: ρ'_i	10	11	13	23	6	17	25	18
4x32: ρ''_i	26	21	27	5	20	11	10	20

3.3 Threefry Family of Block Ciphers

Threefry is a family of block ciphers derived from the well-known Threefish cipher. As with Threefish, Threefry is an ARX cipher.

Each instance is denoted by Threefry–N×W–R, where N is the number of words in the state, W is the bit size of each word, and R is the number of rounds. The recommended and used parameters for Threefry are $\mathrm{N} \in \{2, 4\}$ and $\mathrm{W} \in \{32, 64\}$. Accordingly, the block size of each variant is N × W bits, and also the key size is N × W bits. Figure 3 depicts the round functions of Threefry–2×W (left) and Threefry–4×W (right).

The round function of Threefry–2×W operates on a 2W-bit state $(X^{(i)}, Y^{(i)}) \in \mathbb{F}_2^{\mathrm{W}\times 2}$ as follows:

$$X^{(i+1)} = X^{(i)} + Y^{(i)}\,, \qquad Y^{(i+1)} = X^{(i+1)} \oplus \left(Y^{(i)} \lll \rho_{i \bmod 8}\right),$$

where $\rho_{i \bmod 8}$ is the rotation constant given in Table 2.

The round function of Threefry–4×W operates on a 4W-bit state $(X^{(i)}, Y^{(i)}, Z^{(i)}, T^{(i)}) \in \mathbb{F}_2^{\mathrm{W}\times 4}$, producing the next state $(X^{(i+1)}, Y^{(i+1)}, Z^{(i+1)}, T^{(i+1)})$ as follows:

$$\begin{aligned} X^{(i+1)} &= X^{(i)} + Y^{(i)}\,, & Y^{(i+1)} &= Z^{(i+1)} \oplus \left(T^{(i)} \lll \rho'_{i \bmod 8}\right), \\ Z^{(i+1)} &= Z^{(i)} + T^{(i)}\,, & T^{(i+1)} &= X^{(i+1)} \oplus \left(Y^{(i)} \lll \rho''_{i \bmod 8}\right). \end{aligned}$$

Here, (ρ'_i, ρ''_i) are denoting the rotation constants specified in Table 2. Note that all additions are taken modulo 2^{W}.

The key schedule of Threefry follows that of Threefish, but is simplified by omitting the *tweak* component, which is considered unnecessary for the PRNG use case on GPU platforms. Threefry–N uses N subkey words of W bits: $K_0, \ldots, K_{\mathrm{N}-1}$. The key schedule then derives an additional key word as

$$K_{\mathrm{N}} = K_0 \oplus K_1 \oplus \cdots \oplus K_{\mathrm{N}-1} \oplus C\,,$$

where $C =$ `0x1BD11BDA` for $\mathrm{W} = 32$ and $C =$ `0x1BD11BDAA9FC1A22` for $\mathrm{W} = 64$.

Then, at the input of every fourth round, i.e., the round with index $4s$, the N consecutive subkeys $K_s, K_{s+1}, \ldots$ are added to the cipher state using modular addition. Specifically, for Threefry–2×W, the subkeys $(K_s, K_{s+1} + s)$ are used, and for Threefry–4×W,

the subkeys $(K_s, K_{s+1}, K_{s+2}, K_{s+3} + s)$ are employed, where the indices are taken modulo $\mathrm{N} + 1$.

This implies that the first key addition uses (K_0, K_1) or (K_0, K_1, K_2, K_3) to the plaintext words. Finally, after R rounds (with R being a multiple of 4), the ciphertext is obtained by performing one additional key addition to the final state with $(K_{R/4}, K_{R/4+1} + R/4)$ or $(K_{R/4}, K_{R/4+1}, K_{R/4+2}, K_{R/4+3} + R/4)$.

In the original paper [SMDS11], the authors report that Threefry–2×W and Threefry–4×W achieve full crush-resistance with 13 and 12 rounds, respectively. However, they recommend 20-round variants for both Threefry–2×W and Threefry–4×W in order to provide an additional security margin.

4 Cryptographic Analysis of Philox

In this section, we present the security analysis of Philox ciphers against multiplicative–differential and differential-linear cryptanalysis. We begin by introducing basic concepts on multiplicative differentials; by applying these concepts, we then derive distinguishers for Philox. Then, we convert the Philox–4×W distinguishers into practical key-recovery attacks on 7-, 9-, 10-round Philox–4×W with practical time and data complexity.

4.1 On Multiplicative Differentials

Classical differential and linear cryptanalysis typically rely on the propagation of XOR-based differences and linear masks through the round operations of a cipher. Although these techniques are powerful for operations such as modular addition and XOR, they become much less effective in the presence of modular multiplication.

To address this limitation, Borisov *et al.* [BCJW02] introduced the concept of multiplicative differentials, which replaces the notion of additive differences (including XOR-based differences) with multiplicative ratios. Here, instead of analyzing the evolution of additive differences, one studies how input-output ratios propagate through modular multiplication. This approach preserves an exploitable structure where classical differentials fail, and it has become an established tool for analyzing ciphers that employ multiplication.

They were first exploited against variants of the IDEA block cipher, whose security relies heavily on modular multiplication. Related ideas have also been used against other primitives that combine integer multiplication and XOR, such as Nimbus [Fur01].

In the following, we summarize several structural properties of modular multiplication that will serve as the basis for constructing (multiplicative) differential-linear trails in the analysis of Philox block ciphers.

Based on the distributivity of multiplication over additive inverses, we have

$$\mathrm{mullo}_M(-X) \equiv -\,\mathrm{mullo}_M(X) \bmod 2^{\mathrm{W}}\,.$$

Using this property, for any M, if the inputs differ by a multiplicative factor $-1 \bmod 2^{\mathrm{W}}$, i.e., the pair $(X, -X \bmod 2^{\mathrm{W}})$, then their images under mullo_M also differ by $-1 \bmod 2^{\mathrm{W}}$. In particular, the outputs form the pair $(Y, -Y \bmod 2^{\mathrm{W}})$ with $Y = \mathrm{mullo}_M(X)$. Thus, multiplicative differentials are preserved with probability 1 through the multiplication operations.

Lemma 1. *Let X be an odd integer in $\mathbb{Z}_{2^{\mathrm{W}}}$. Then,*

$$-X \bmod 2^{\mathrm{W}} \equiv X \oplus (2^{\mathrm{W}} - 2)\,.$$

Lemma 2. *Let M be an odd integer in $\mathbb{Z}_{2^{\mathrm{W}}}$. Then, for any non-zero $X \in \mathbb{Z}_{2^{\mathrm{W}}}$, we have*

$$\left\lfloor \frac{X \times M}{2^{\mathrm{W}}} \right\rfloor + \left\lfloor \frac{(-X) \times M}{2^{\mathrm{W}}} \right\rfloor = M - 1\,.$$

The proofs of the above lemmas can be found in previous works, such as [BCJW02], and are therefore omitted here.

Using Lemma 2 and translating it into PHILOX operations, we obtain the following: For any non-zero $X \in \mathbb{F}_2^{\mathrm{W}}$

$$\mathrm{mulhi}_M(-X) \equiv -\mathrm{mulhi}_M(X) + M - 1 \bmod 2^{\mathrm{W}}.$$

Based on these lemmas, we construct a series of deterministic or highly correlated (multiplicative) differential-linear distinguishers on PHILOX−2×W and PHILOX−4×W. We present the specific constructions and complete validation proofs for the distinguishers in Appendix B. In the following subsection, we focus on a six round distinguisher for PHILOX−4×W that we use to mount our key-recovery attacks.

4.2 6-Round Distinguishers on Philox−4×W

We present a (multiplicative) differential-linear distinguisher on PHILOX−4×W reduced to six rounds that optimizes the adversarial data-collection model while maintaining a deterministic advantage. This distinguisher represents a strategic compromise: It retains a distinguishing probability of exactly 1 while relaxing the rigid plaintext structure requirements of our first construction. This structural flexibility makes it exceptionally well-suited for mounting practical attacks on counter-based random number generators.

Distinguisher 1. *Let* $P = (X^{(0)}, Y^{(0)}, Z^{(0)}, T^{(0)})$ *and* $\tilde{P} = (\tilde{X}^{(0)}, \tilde{Y}^{(0)}, \tilde{Z}^{(0)}, \tilde{T}^{(0)})$ *be two plaintexts for* PHILOX−4×W*, with* $X_0^{(0)} = \tilde{X}_0^{(0)}$ *(resp.* $Z_0^{(0)} = \tilde{Z}_0^{(0)}$*), and*

$$\begin{aligned}
&\tilde{Y}^{(0)} = Y^{(0)}, \quad \tilde{Z}^{(0)} = Z^{(0)},\\
&\tilde{T}^{(0)} = T^{(0)} \oplus \mathrm{mulhi}_{M_0}(X^{(0)}) \oplus \mathrm{mulhi}_{M_0}(\tilde{X}^{(0)})\\
\Big(\textit{resp. } &\tilde{X}^{(0)} = X^{(0)}, \quad \tilde{T}^{(0)} = T^{(0)},\\
&\tilde{Y}^{(0)} = Y^{(0)} \oplus \mathrm{mulhi}_{M_1}(Z^{(0)}) \oplus \mathrm{mulhi}_{M_1}(\tilde{Z}^{(0)})\Big).
\end{aligned}$$

Then, the corresponding states after five rounds, i.e., $(X^{(5)}, Y^{(5)}, Z^{(5)}, T^{(5)})$ *and* $(\tilde{X}^{(5)}, \tilde{Y}^{(5)}, \tilde{Z}^{(5)}, \tilde{T}^{(5)})$*, satisfy the following relation:*

$$T_0^{(5)} = \tilde{T}_0^{(5)} \quad \left(\textit{resp. } Y_0^{(5)} = \tilde{Y}_0^{(5)}\right).$$

Besides, the corresponding states after six rounds, i.e., $(X^{(6)}, Y^{(6)}, Z^{(6)}, T^{(6)})$ *and* $(\tilde{X}^{(6)}, \tilde{Y}^{(6)}, \tilde{Z}^{(6)}, \tilde{T}^{(6)})$*, satisfy the following relation:*

$$\begin{aligned}
\Pr\left[\left(F_{M_0}(T^{(6)}) \oplus Z^{(6)}\right)_0 = \left(F_{M_0}(\tilde{T}^{(6)}) \oplus \tilde{Z}^{(6)}\right)_0\right] &= 1,\\
\Big(\textit{resp. } \Pr\left[\left(F_{M_1}(Y^{(6)}) \oplus X^{(6)}\right)_0 = \left(F_{M_1}(\tilde{Y}^{(6)}) \oplus \tilde{X}^{(6)}\right)_0\right] &= 1\Big),
\end{aligned}$$

where $F_{M_i} := \mathrm{mulhi}_{M_i} \circ \mathrm{mullo}_{M_i}^{-1}$ *for* $i \in \{0, 1\}$.

Proof. To prove this distinguisher, we analyze the difference propagation through each round of PHILOX−4×W. Since $\tilde{Y}^{(0)} = Y^{(0)}$ and $\tilde{Z}^{(0)} = Z^{(0)}$, it follows directly that $\tilde{X}^{(1)} = X^{(1)}$ and $\tilde{Y}^{(1)} = Y^{(1)}$. For $Z^{(1)}$ and $\tilde{Z}^{(1)}$, we compute:

$$\begin{aligned}
Z^{(1)} &= T^{(0)} \oplus \mathrm{mulhi}_{M_0}(X^{(0)}) \oplus K_1^{(0)},\\
\tilde{Z}^{(1)} &= \tilde{T}^{(0)} \oplus \mathrm{mulhi}_{M_0}(\tilde{X}^{(0)}) \oplus K_1^{(0)}\\
&= T^{(0)} \oplus \mathrm{mulhi}_{M_0}(X^{(0)}) \oplus \mathrm{mulhi}_{M_0}(\tilde{X}^{(0)}) \oplus \mathrm{mulhi}_{M_0}(\tilde{X}^{(0)}) \oplus K_1^{(0)}\\
&= T^{(0)} \oplus \mathrm{mulhi}_{M_0}(X^{(0)}) \oplus K_1^{(0)} = Z^{(1)}.
\end{aligned}$$

Moreover, since M_0 is odd and $X_0^{(0)} = \tilde{X}_0^{(0)}$, we obtain $\tilde{T}_0^{(1)} = T_0^{(1)}$. Hence, we establish the first-round output relations:

$$\tilde{X}^{(1)} = X^{(1)}\,, \quad \tilde{Y}^{(1)} = Y^{(1)}\,, \quad \tilde{Z}^{(1)} = Z^{(1)}\,, \quad \tilde{T}_0^{(1)} = T_0^{(1)}\,.$$

This deterministic behavior continues into the second round, where we find:

$$\tilde{X}^{(2)} = X^{(2)}\,, \quad \tilde{Y}^{(2)} = Y^{(2)}\,, \quad \tilde{Z}_0^{(2)} = Z_0^{(2)}\,, \quad \tilde{T}^{(2)} = T^{(2)}\,.$$

Similarly, since M_1 is odd, the invariant conditions after the third round become:

$$\tilde{Y}_0^{(3)} = Y_0^{(3)}\,, \quad \tilde{Z}^{(3)} = Z^{(3)}\,, \quad \tilde{T}^{(3)} = T^{(3)}\,.$$

Therefore, after evaluating the fourth round, it follows that:

$$\tilde{X}_0^{(4)} = X_0^{(4)}\,, \quad \tilde{Y}^{(4)} = Y^{(4)}\,.$$

Finally, since M_0 is odd, the fifth round update yields a matching parity in the target branch:

$$\tilde{T}_0^{(5)} = T_0^{(5)}\,.$$

The above 5-round differential-linear distinguisher can be extended by one round, by using the fact that $\mathrm{mullo}_{M_i}(\cdot)$ is a bijection. From the round update structural equations, we obtain

$$\begin{aligned} T^{(5)} &= \mathrm{mulhi}_{M_0}(X^{(5)}) \oplus Z^{(6)} \oplus K_1^{(5)} \\ &= \mathrm{mulhi}_{M_0}\left(\mathrm{mullo}_{M_0}^{-1}(T^{(6)})\right) \oplus Z^{(6)} \oplus K_1^{(5)} \\ &= F_{M_0}(T^{(6)}) \oplus Z^{(6)} \oplus K_1^{(5)}\,. \end{aligned}$$

Consequently, the parity preservation condition translates directly to

$$\left(F_{M_0}(\tilde{T}^{(6)}) \oplus \tilde{Z}^{(6)}\right)_0 = \left(F_{M_0}(T^{(6)}) \oplus Z^{(6)}\right)_0\,.$$

Thus, the distinguisher naturally extends to a 6-round differential-linear distinguisher holding with probability 1. □ □

The primary advantage of Distinguisher 1 lies in its alignment with standard counter-based random number generators. The plaintext difference here is strictly localized to the lowest-order counter branch, $T^{(0)}$. Moreover, an attacker can select the remaining inputs to ensure that the higher-order word multiplication difference $\mathrm{mulhi}_{M_0}(X^{(0)}) \oplus \mathrm{mulhi}_{M_0}(\tilde{X}^{(0)})$ is minimal, such as a single-bit difference. Specifically, by fixing the three left branches and selecting odd values for $X^{(0)}$ and $\tilde{X}^{(0)}$ that satisfy $\mathrm{mulhi}_{M_0}(X^{(0)}) \oplus \mathrm{mulhi}_{M_0}(\tilde{X}^{(0)}) = 1$, the structural requirements of the distinguisher are naturally fulfilled alongside a single-bit difference in the LSB of the T branch. In other words, within at most $2^{\mathrm{W}+1}$ executions of the generator, a large number of suitable plaintext pairs can be efficiently obtained.

4.3 Finding Distinguishers with AlphaEvolve

To identify distinguishers spanning a higher number of rounds (or with a higher bias), we used AlphaEvolve [NVE+25], an evolutionary coding agent. AlphaEvolve uses LLM models to iteratively generate programs, scores them against specified metrics, and feeds the evaluation results back to the model to guide subsequent generations. This simple approach has shown broad applicability and already could improve the state-of-the-art on many problems across different domains.

The architecture of our experiment to find distinguishers for Philox–4×W is outlined in Figure 4, and we use the following setup:

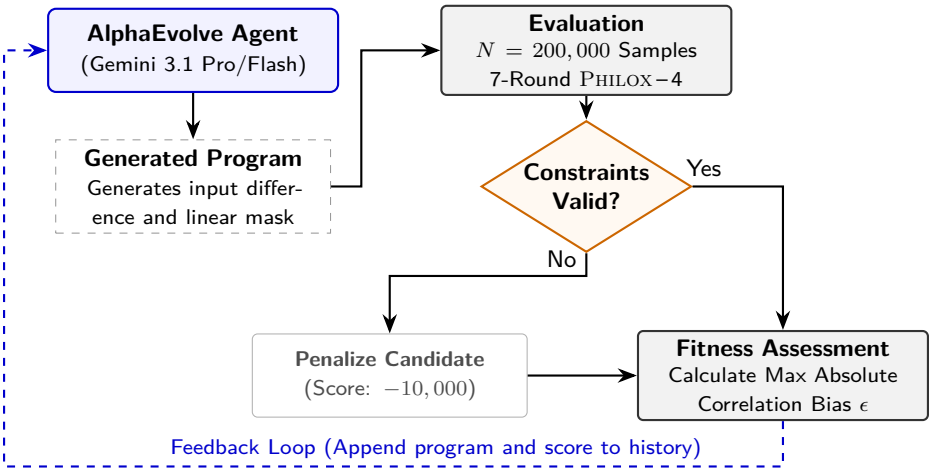


Figure 4: Architecture of the AlphaEvolve pipeline used to find the 8-round distinguisher for PHILOX−4×W. Note that we search for a practical 7-round distinguisher, which can be manually extended to 8 rounds.

- Problem Context: We ask an LLM to generate a program which outputs a candidate for an (multiplicative) input difference and output mask for PHILOX−4×W (full prompt is given in Appendix F).
- Evaluation: We experimentally evaluate the bias on the provided output mask for 200.000 samples. The score of a candidate program is then set to the maximum absolute bias.

We set the target to find a 7-round distinguisher to improve upon our manual cryptanalysis. AlphaEvolve can consistently find novel solutions with a complex input difference pattern which yields a very high bias for 7 rounds after a small number of program candidates. This typically requires less than 100 programs to be generated before AlphaEvolve (using Gemini-3.1-pro and Gemini-3.1-flash as the underlying LLM models) generates a program which outputs the distinguisher below.

As the distinguisher we find uses a simple bit-mask in a single word at the output, we can use the structural observations made in the previous sections to unwind parts of the last round and extend the distinguisher to 8 rounds.

Distinguisher 2. *Let* $P = (X^{(0)}, Y^{(0)}, Z^{(0)}, T^{(0)})$ *and* $\tilde{P} = (\tilde{X}^{(0)}, \tilde{Y}^{(0)}, \tilde{Z}^{(0)}, \tilde{T}^{(0)})$ *be two plaintexts for* PHILOX−4×W *such that*

$$\tilde{X}^{(0)} = X^{(0)}, \quad \tilde{Y}^{(0)} = Y^{(0)} \oplus \mathrm{mulhi}_{M_1}(Z^{(0)}) \oplus \mathrm{mulhi}_{M_1}(\tilde{Z}^{(0)}),$$
$$\tilde{Z}^{(0)} = \mathrm{mullo}_{M_1^{-1}}(\mathrm{mullo}_{M_1}(Z^{(0)}) \oplus 1), \quad \textit{and} \quad \tilde{T}^{(0)} = T^{(0)}.$$

Then the corresponding states after eight rounds satisfy

$$\Pr\left[\left(F_{M_0}(T^{(8)}) \oplus Z^{(8)}\right)_1 = \left(F_{M_0}(\tilde{T}^{(8)}) \oplus \tilde{Z}^{(8)}\right)_1\right] \approx 0.66.$$

We analyze the first-round difference propagation to explain why the automated search yields such a strong bias over 8 rounds. In the initial state, non-zero differences appear only in the Y and Z branches. The difference between $Z^{(0)}$ and $\tilde{Z}^0$ is chosen such that a difference of 1 occurs after applying mullo_{M_1}. The complex difference which propagates to

mulhi_{M_1} and is XORed with the Y branch is perfectly canceled out with the input difference chosen for the Y branch. This means after one round the difference between the two states is strictly $(0, 1, 0, 0)$, which is equivalent to the input difference for Distinguisher 6. This immediately allows us to improve Distinguisher 6 as the bit-mask found by AlphaEvolve has a significantly higher bias, than the LSB one we manually derived.

We were unable to extend these automated results to 9 rounds. Using the same experimental setup to search for an 8-round distinguisher targeting a simple bit-mask – intended for a subsequent 9-round extension – did not produce any high probability distinguishers even after > 100.000 program generations.

Finally, we observed that the statistical tests in TestU01 is sensitive to the effects of Distinguisher 2 and Philox−4×W cannot achieve full crush-resistance with 7 rounds as the designers claimed. Designers considered two adversarial counters where they considered *gray-coded* and *constant Hamming counters*. However, for a random plaintext and key, we considered plaintext pairs as provided in Distinguisher 2 and incremented the plaintext by one. Concatenated 7-round ciphertexts failed even the Crush Tests in which 8 tests failed on average and for the BigCrush tests, more than 15 tests failed. *Crush-resistance* is obtained when such plaintexts are encrypted for 8 rounds.

4.4 Key Recovery Attack on 7-, 9-, and 10-Round Philox−4×W

The 6-round distinguishers on Philox−4×W can be extended to a key-recovery attack on the 7-round variant by appending one additional round at the end. Note that employing any of the proposed distinguishers leads to an analogous key-recovery procedure. Therefore, we focus our description on the attack using Distinguisher 6, adapted to the improved output mask discovered by AlphaEvolve.

7-Round Attack using Distinguisher 6.

Consider a pair of plaintexts $P = (X^{(0)}, Y^{(0)}, Z^{(0)}, T^{(0)})$ and $\tilde{P} = (\tilde{X}^{(0)}, \tilde{Y}^{(0)}, \tilde{Z}^{(0)}, \tilde{T}^{(0)})$ for Philox−4×W such that

$$\tilde{X}^{(0)} = X^{(0)}\,, \quad \tilde{Y}^{(0)} = Y^{(0)}\,, \quad \tilde{Z}^{(0)} = Z^{(0)}\,, \quad \tilde{T}^{(0)} = T^{(0)} \oplus 1\,.$$

Then it is required to check that the corresponding internal states after six rounds, $(X^{(6)}, Y^{(6)}, Z^{(6)}, T^{(6)})$ and $(\tilde{X}^{(6)}, \tilde{Y}^{(6)}, \tilde{Z}^{(6)}, \tilde{T}^{(6)})$, satisfy the following biased relation on bit index 1:

$$\Pr\left[\left(F_{M_0}(\tilde{T}^{(6)}) \oplus \tilde{Z}^{(6)}\right)_1 = \left(F_{M_0}(T^{(6)}) \oplus Z^{(6)}\right)_1\right] \approx 0.66\,.$$

By applying the inverse relations for the final round, we obtain the following:

$$\begin{cases} Z^{(6)} = \text{mullo}_{M_1}^{-1}(Y^{(7)}) \\ \tilde{Z}^{(6)} = \text{mullo}_{M_1}^{-1}(\tilde{Y}^{(7)}) \end{cases} \quad \text{and,} \quad \begin{cases} T^{(6)} = F_{M_0}(T^{(7)}) \oplus Z^{(7)} \oplus K_1^{(6)} \\ \tilde{T}^{(6)} = F_{M_0}(\tilde{T}^{(7)}) \oplus \tilde{Z}^{(7)} \oplus K_1^{(6)} \end{cases}$$

where $K_1^{(6)}$ is the relevant subkey added in this round. Substituting these expressions into the distinguisher condition yields the following constraint on the 7-round states and the unknown subkey:

$$\Pr\Big[\Big(F_{M_0}\big(F_{M_0}(\tilde{T}^{(7)}) \oplus \tilde{Z}^{(7)} \oplus K_1^{(6)}\big) \oplus \text{mullo}_{M_1}^{-1}(\tilde{Y}^{(7)})\Big)_1 = \Big(F_{M_0}\big(F_{M_0}(T^{(7)}) \oplus Z^{(7)} \oplus K_1^{(6)}\big) \oplus \text{mullo}_{M_1}^{-1}(Y^{(7)})\Big)_1\Big] \approx 0.66\,.$$

This relation serves as the key-filtering equation for our attack and depends on the unknown subkey word $K_1^{(6)}$. Importantly, in the counter-based PRNG setting this gives a

ciphertext-only attack: the adversary does not need chosen plaintext-ciphertext pairs, since the inputs to PHILOX are consecutive counter values and their difference pattern (i.e. difference only in LSB) is known from the structure of the generator. Given enough consecutive pairs of observed PRNG outputs, an adversary can guess a candidate for $K_1^{(6)}$ and evaluate the key-filtering equation. For the correct guess, the equality holds with probability 66%, whereas for an incorrect guess, it holds with an expected probability of 50%.

The attack requires evaluating N independent output pairs, counting how often the equation holds for each candidate guess. The correct key exhibits a bias $\epsilon = p - 1/2$, allowing us to distinguish it from wrong guesses using standard statistical hypothesis testing. The number of required pairs is proportional to ϵ^{-2}. Following a standard rule of thumb, this number is multiplied by the number of guessed bits. Thus, for Distinguisher 6, the attack requires $N \approx \mathrm{W} \cdot (0.16)^{-2} \approx \mathrm{W} \cdot 2^{5.29}$ pairs. This yields a total data complexity of $\mathrm{W} \cdot 2^{6.29}$ observed PRNG outputs, ensuring that the correct key is ranked first by its bias with high probability. Finally, once $K_1^{(6)}$ is recovered, the original base key is obtained via $K_1 = K_1^{(6)} - 6 \cdot C_1 \bmod 2^{\mathrm{W}}$.

Regarding the time complexity, we collect $N \approx \mathrm{W} \cdot 2^{5.29}$ output pairs. For each of 2^{W} possible candidate values of the subkey $K_1^{(6)}$, we evaluate the key-filtering equation over all N pairs. This approach requires approximately $N \cdot 2^{\mathrm{W}}$ partial decryptions. Each partial decryption for a given pair and a specific guess of $K_1^{(6)}$ requires four evaluations of F_{M_0} and two of $\mathrm{mullo}_{M_1}^{-1}$. However, two of the F_{M_0} evaluations and both $\mathrm{mullo}_{M_1}^{-1}$ operations depend solely on the observed outputs and are independent of the subkey guess. By performing a straightforward precomputation on the output pairs, the computational cost per subkey guess is reduced to only two evaluations of F_{M_0}. Since two evaluations of F_{M_0} are approximately equivalent to one round of PHILOX$-4\times$W, the time complexity to recover $K_1^{(6)}$ is evaluated as:

$$T_{K_1} \approx \frac{N \cdot 2^{\mathrm{W}} \cdot 1}{7} \approx \frac{\mathrm{W} \cdot 2^{5.29} \cdot 2^{\mathrm{W}}}{7} \approx \mathrm{W} \cdot 2^{\mathrm{W}+2.48}$$

PHILOX$-4\times$W-7 encryptions. Once K_1 is recovered, the remaining key word K_0 can be determined via an exhaustive search with a cost of $T_{K_0} \approx 2^{\mathrm{W}}$ encryptions. Hence, the overall time complexity is:

$$T \approx T_{K_1} + T_{K_0} \approx \mathrm{W} \cdot 2^{\mathrm{W}+2.48} + 2^{\mathrm{W}} \approx \mathrm{W} \cdot 2^{\mathrm{W}+2.48}$$

PHILOX$-4\times$W-7 encryptions, which is significantly faster than a standard brute-force search requiring $2^{2\mathrm{W}}$ encryptions.

7-Round Attack using Distinguisher 1.

The primary difference in the key-recovery procedure between the attacks using Distinguisher 6 and Distinguisher 1 is that, although the key-filtering equation retains the same structural form, the correlation becomes perfect for the correct key guess in the latter case. In particular, for the correct guess, the equality holds with probability 1, whereas for an incorrect guess, it holds with a probability of 0.5. Hence, if we evaluate d independently chosen pairs, an incorrect key candidate survives all d tests with a probability of 2^{-d}, which is sufficient to uniquely isolate the correct key. Thus, the naive computational cost of this verification stage is approximately $T_{\text{naive}} \approx 2d \cdot 2^{\mathrm{W}}$ single-round operations of PHILOX$-4\times$W.

However, these pairs can be optimized by choosing a single random base input P_0 and then selecting d distinct inputs P_i for $1 \le i \le d$, such that each pair (P_0, P_i) satisfies the input pattern of the distinguisher. In this way, the required data complexity reduces

to $d+1$ inputs instead of $2d$. Consequently, the total time complexity is approximately $T \approx (d+1) \cdot 2^{\mathrm{W}} \cdot (1/7)$ 7-round Philox$-4\times$W encryptions.

The key-recovery attacks based on Distinguisher 5 and Distinguisher 1 require a relatively small number of chosen inputs, which can be collected within at most $2^{\mathrm{W}+1}$ executions of the counter-based PRNG. Although the attack based on Distinguisher 6 features a slightly higher time complexity, it represents the most practical threat model and perfectly aligns with the counter-based PRNG scenario. In particular, it is ciphertext-only and requires only about $2,500$ *consecutive* PRNG outputs, which is an exceptionally small window in real-world applications. For instance, as illustrated in Figure 1, standard tasks like image generation typically involve more than 10^5 calls to the underlying PRNG. Hence, the required number of samples is well within the bounds of normal operational usage. Crucially, the adversary does not require any control over the seed or counter inputs, since the target input pattern naturally emerges during standard PRNG operation.

9-Round Attack using Distinguisher 2.

Since the 8-round distinguisher in Distinguisher 2 shares an identical final-round structure with Distinguisher 6, it can be utilized directly to mount a 9-round key-recovery attack. As a result, this attack inherits the exact same data and time complexities as the 7-round attack using Distinguisher 6.

10-Round Attack using Distinguisher 2.

To mount a 10-round key-recovery attack, we apply the 8-round distinguisher from Distinguisher 2 by appending one round before and one round after it for key-recovery purpose. Accordingly, we need enough pairs of intermediate states $(X^{(1)}, Y^{(1)}, Z^{(1)}, T^{(1)})$ and $(\tilde{X}^{(1)}, \tilde{Y}^{(1)}, \tilde{Z}^{(1)}, \tilde{T}^{(1)})$ that satisfy the following difference pattern:

$$\tilde{X}^{(1)} = X^{(1)}\,, \quad \tilde{T}^{(1)} = T^{(1)}\,, \quad \tilde{Z}^{(1)} = \mathrm{mullo}_{M_1^{-1}}(\mathrm{mullo}_{M_1}(Z^{(1)}) \oplus 1)\,,$$
$$\tilde{Y}^{(1)} = Y^{(1)} \oplus \mathrm{mulhi}_{M_1}(Z^{(1)}) \oplus \mathrm{mulhi}_{M_1}(\tilde{Z}^{(1)})\,.$$

Consider a pair of plaintexts $P = (X^{(0)}, Y^{(0)}, Z^{(0)}, T^{(0)})$ and $\tilde{P} = (\tilde{X}^{(0)}, \tilde{Y}^{(0)}, \tilde{Z}^{(0)}, \tilde{T}^{(0)})$ for Philox$-4\times$W that satisfies these target relations at the intermediate state after one round. By inverting the first-round function, the plaintext components must satisfy the following conditions:

$$\tilde{X}^{(0)} = X^{(0)}\,, \quad \tilde{Y}^{(0)} = Y^{(0)} \oplus \mathrm{mulhi}_{M_1}(\tilde{Z}^{(0)}) \oplus \mathrm{mulhi}_{M_1}(Z^{(0)})\,,$$
$$\tilde{Z}^{(0)} = \mathrm{mullo}_{M_1^{-1}}\left(\mathrm{mullo}_{M_1}(Z^{(0)}) \oplus \mathrm{mulhi}_{M_1}(Z^{(1)}) \oplus \mathrm{mulhi}_{M_1}(\tilde{Z}^{(1)})\right),$$

along with the difference relation on the remaining branch:

$$\mathrm{mullo}_{M_0}\left(\mathrm{mulhi}_{M_0}(\tilde{T}^{(0)}) \oplus X^{(0)} \oplus K_1^{(0)}\right) \oplus \mathrm{mullo}_{M_0}\left(\mathrm{mulhi}_{M_0}(T^{(0)}) \oplus X^{(0)} \oplus K_1^{(0)}\right) = 1\,.$$

Consequently, guessing the subkey word $K_1^{(0)} = K_1$ allows the adversary to construct plaintext pairs that guarantee the internal state values at the beginning of the distinguisher fulfill the required conditions. Furthermore, similar to the 7-round attack, we evaluate the key-filtering equation at the output of the tenth round to check the biased relation on bit index 1:

$$\Pr\left[\left(F_{M_0}\left(F_{M_0}(\tilde{T}^{(10)}) \oplus \tilde{Z}^{(10)} \oplus K_1^{(9)}\right) \oplus \mathrm{mullo}_{M_1}^{-1}(\tilde{Y}^{(10)})\right)_1 = \left(F_{M_0}\left(F_{M_0}(T^{(10)}) \oplus Z^{(10)} \oplus K_1^{(9)}\right) \oplus \mathrm{mullo}_{M_1}^{-1}(Y^{(10)})\right)_1\right] \approx 0.66\,,$$

Table 3: Complexity of key recovery attacks on PHILOX–4×W. Note that for Distinguisher 6 we use the improved output mask found by AlphaEvolve.

Cipher	**Distinguisher**	**Data** [CP]	**Time** [Enc]
PHILOX–4×32–7	Distinguisher 1	$2^{5.04}$	$2^{34.23}$
	Distinguisher 5	$2^{5.04}$	$2^{34.23}$
	Distinguisher 6	$2^{11.29}$CC	$2^{39.48}$
PHILOX–4×32–9	Distinguisher 2	$2^{11.29}$	$2^{39.48}$
PHILOX–4×32–10	Distinguisher 2	$2^{43.29}$	$2^{43.46}$
PHILOX–4×64–7	Distinguisher 1	$2^{6.02}$	$2^{67.23}$
	Distinguisher 5	$2^{6.02}$	$2^{67.23}$
	Distinguisher 6	$2^{12.29}$	$2^{72.48}$

which requires knowledge of the subkey $K_1^{(9)} = K_1 + 9 \cdot C_1 \pmod{2^{\mathrm{W}}}$. Therefore, by combining these two extensions, the adversary only needs to guess the single subkey word K_1, which simultaneously enables both the generation of valid plaintext pairs and the verification of the bias at the ciphertext output.

To mount the attack, the adversary first fixes a value for $Z^{(1)}$, which uniquely determines the value for $\tilde{Z}^{(1)}$. This choice determines the differences $\alpha_1 = Z^{(1)} \oplus \tilde{Z}^{(1)} = T^{(0)} \oplus \tilde{T}^{(0)}$ and $\beta_1 = Y^{(1)} \oplus \tilde{Y}^{(1)}$. Next, the adversary fixes the plaintext word $Z^{(0)}$ to a constant value z, which subsequently determines $\tilde{Z}^{(0)}$ along with the plaintext differences $\alpha_0 = Z^{(0)} \oplus \tilde{Z}^{(0)}$ and $\beta_0 = Y^{(0)} \oplus \tilde{Y}^{(0)}$. The plaintext word $X^{(0)}$ is likewise fixed to a constant value x.

For each of the 2^{W} possible values of t assigned to the branch $T^{(0)}$, the adversary randomly selects N distinct values for y to populate the $Y^{(0)}$ branch. The adversary then queries the encryption oracle for the plaintext pairs (x, y, z, t) and $(x, y \oplus \beta_0, z \oplus \alpha_0, t \oplus \alpha_1)$. For each chosen value of t, a unique candidate is determined via $K_1 = t \oplus Z^{(1)} \oplus \mathrm{mulhi}_{M_0}(x)$, which also determines the corresponding final-round subkey $K_1^{(9)} = K_1 + 9 \cdot C_1 \pmod{2^{\mathrm{W}}}$. Using these values, the adversary evaluates the key-filtering equation over the N collected plaintext–ciphertext pairs. If the subkey guess is correct, the bias will match the expected value. For any incorrect subkey guess, the filtering property behaves randomly and holds with a probability of 0.5, causing the computed bias to 0.

As explained previously in the 7-round attack, the verification phase requires $N = \mathrm{W} \cdot 2^{5.29}$ pairs per subkey. Since this data collection is repeated across all 2^{W} values of t, the data complexity of the attack is $\mathrm{W} \cdot 2^{\mathrm{W}+6.29}$ chosen plaintexts.

Regarding the time complexity, the adversary must first perform $\mathrm{W} \cdot 2^{\mathrm{W}+6.29}$ encryption queries. The key-filtering equation is evaluated over the N pairs for each of the 2^{W} subkey. The computational cost per pair is four evaluations of F_{M_0} and two of $\mathrm{mullo}_{M_1}^{-1}$, which is equivalent to 2.5 rounds of PHILOX–4×W. Thus, the time complexity required to recover K_1 is evaluated as:

$$2^{\mathrm{W}} \cdot N \cdot (2 + \frac{2.5}{10}) \approx 2^{\mathrm{W}} \cdot \mathrm{W} \cdot 2^{5.29} \cdot 2^{1.17} \approx \mathrm{W} \cdot 2^{\mathrm{W}+6.46}$$

10-round PHILOX–4×W encryptions. Once K_1 is successfully recovered, the key word K_0 can be determined via a standard exhaustive search. Consequently, the overall time complexity of the 10-round attack is $T \approx \mathrm{W} \cdot 2^{\mathrm{W}+6.46}$ encryptions, which remains significantly faster than an exhaustive brute-force search over the $2^{2\mathrm{W}}$ key space. We summarize the data and time complexities of our proposed key-recovery attacks on PHILOX–4×W in Table 3.

4.5 Experimental Verification.

All distinguishers and key-recovery attacks on PHILOX−4×32−7 were fully implemented and are provided in the supplementary code. The key-recovery attacks based on Distinguishers 5 and 1 take approximately five minutes on a standard laptop equipped with an Intel Core Ultra 7 CPU and 16 GB of RAM. The attacks on 7 and 9 rounds, based on Distinguisher 6 and Distinguisher 2, respectively, require about 30 minutes to recover the key on the same machine.

One may ask whether Distinguisher 3 or Distinguisher 4 can be used within a key-recovery attack on PHILOX−2. Indeed, the 5-round distinguishers can be turned into key-recovery attacks against 6-round PHILOX−2. However, in this setting, the attacker must guess the single W-bit key word used by the cipher, which amounts to searching the *entire* key space. Consequently, any advantage of this key recovery attack over a straightforward brute force search is negligible. For this reason, we omit a detailed description of key recovery on PHILOX−2.

5 Cryptographic Analysis of Reduced-Round Threefry

In this section, we propose a new generic method for searching for differential-linear distinguishers, and apply it to THREEFRY. We first explain why we need the new method. Then, we describe the segmented search algorithm. Finally, we present our results on THREEFRY−2×32.

5.1 Differential-Linear Distinguishers

We decompose E into three segments $E = E_\ell \circ E_m \circ E_u$, where E_u, E_m, and E_ℓ consist of r_u, r_m, and r_ℓ rounds, respectively. Suppose that the following hold: E_u admits a differential, E_ℓ admits a linear approximation, and E_m admits a combined differential-linear distinguisher, such that:

$$\begin{aligned}
&\Pr_{P,K}(E_u(P) \oplus E_u(P + \Delta_u) = \Delta_m) = p,\\
&\operatorname{Cor}_{X,K}(\langle \Gamma_m, X\rangle \oplus \langle \Gamma_o, E_\ell(X)\rangle) = q,\\
&\operatorname{Cor}_{X,K}(\langle \Gamma_m, E_m(X) \oplus E_m(X + \Delta_m)\rangle) = r.
\end{aligned}$$

Then the correlation of the differential-linear distinguisher for E is approximated by [HDE24, BLN17, BDKW19]:

$$\operatorname{Cor}_{P,K}(\langle \Gamma_o, E(P) \oplus E(P + \Delta_u)\rangle) \approx p \cdot r \cdot q^2.$$

Most existing methods for finding differential-linear distinguishers follow a bottom-up approach, where the cipher is broken down into its components (such as S-boxes, modular addition, XOR), and the propagation of differences and masks through these components is modeled [HDE24, BGG+23]. This process requires a detailed understanding of how to model each component and how to combine them to form the full model of the cipher. It is also prone to modeling errors. In addition, modeling some operations, such as modular addition, can be challenging and often requires approximations that affect the accuracy of the results [BGG+23]. Another limitation is that most existing models assume statistical independence between rounds and components, which may not hold in practice and can lead to large deviations between the estimated and actual probabilities and correlations.

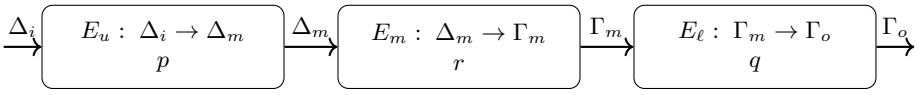


Figure 5: Segmented search for a differential-linear distinguisher. We split E as $E = E_\ell \circ E_m \circ E_u$ and analyze each segment in place, respecting the absolute round boundaries and the key-addition schedule.

5.2 Segmented Search for Differential-Linear Distinguishers

In this work, we follow a top-down approach to find differential-linear distinguishers by segmenting the cipher into three parts and analyzing each part separately. Our method treats the cipher as an oracle and only calls reduced-round encryption and decryption for the segments under analysis. We do not build MILP or CP models and we do not rely on implementation details beyond the round structure and key-addition schedule. This keeps the method simple, portable across designs, and focused on structurally experimental validated results. We explain our method for finding the differential-linear distinguishers of Threefry, but it is general and can also be applied to other primitives.

Segmentation and Method.

As shown in Figure 5, we split the full cipher E into three segments E_u, E_m, and E_ℓ. Recent studies on differential-linear cryptanalysis [HDE24, GWH+25], especially on ARX ciphers, show that many strong differential-linear distinguishers follow an hourglass structure. In this structure, the number of active bits shrinks near the input and output of the middle segment E_m and expands again toward the input and output of the full cipher E. We therefore focus on differences Δ_m and masks Γ_m with low Hamming weight. In particular, we restrict our search to single-bit differences and masks at the junctions between segments (i.e., Δ_m and Γ_m).

For analyzing Threefry, we start at an absolute round `offset` inside the full-design and respect the key schedule method of the cipher. In Threefry, a subkey is added at the beginning and then subkeys are added every four rounds. In our segmentation, we respect these absolute round boundaries and add the subkeys exactly where the design prescribes, so that the combination of the three segments is equivalent to the corresponding reduced-round version of Threefry with a given `offset`. In the following steps, expectations and other statistical measures are computed over the indicated random variables, with K ranging over keys and X over states.

Algorithm Flow.

The method proceeds in five steps, corresponding to the three segments and the final ranking and verification.

1. *Phase M: Searching for a differential-linear distinguisher on the middle segment.* For each pair (Δ_m, Γ_m) with a single active bit in both Δ_m and Γ_m, we estimate the signed correlation r using N random samples:

$$r := \mathbb{E}_{K,X}\left[(-1)^{\langle \Gamma_m, E_{m,K}(X)\rangle \oplus \langle \Gamma_m, E_{m,K}(X \oplus \Delta_m)\rangle}\right].$$

   Here, $\mathbb{E}_{K,X}[Z] = \frac{1}{|\mathcal{K}|\,|\mathcal{X}|}\sum_{K\in\mathcal{K}}\sum_{X\in\mathcal{X}} Z(K,X)$ with K and X sampled uniformly. Equivalently, $\Pr[\langle \Gamma_m, E_{m,K}(X)\rangle = \langle \Gamma_m, E_{m,K}(X \oplus \Delta_m)\rangle] = \frac{1+r}{2}$. We discard pairs (Δ_m, Γ_m) whenever the estimated correlation r falls below a threshold, e.g., $|r| < 2^{-N/2}$, where N denotes the total number of random samples used in the estimation.

By the end of this phase, we store a table of tuples $\texttt{DLCT}[\Delta_m, \Gamma_m] = r$ for the surviving pairs.

2. *Phase U: Searching for a differential distinguisher on the first segment.* For each Δ_m that survives Phase M, we search for an input difference Δ_i that maximizes the probability

$$p = \Pr_{K,Y}\left[E_{u,K}^{-1}(Y) \oplus E_{u,K}^{-1}(Y \oplus \Delta_m) = \Delta_i\right].$$

To this end, we sample many random states Y and form pairs $(Y, Y \oplus \Delta_m)$. We then decrypt these pairs backward through the inverse of E_u to obtain inputs (X, X') at the start of E_u, where $X = E_u^{-1}(Y)$, $X' = E_u^{-1}(Y \oplus \Delta_m)$, and $\Delta_i = X \oplus X'$. We estimate p by counting over many randomly chosen pairs $(Y, Y \oplus \Delta_m)$ and keys K. The decryption process mirrors the composed run and applies inverse subkey additions only at the absolute boundaries where the design prescribes key addition. For each Δ_m, we keep the Δ_i that maximizes the observed probability p. At the end of this phase, we obtain a table of tuples $\texttt{DDT}[\Delta_i, \Delta_m] = p$.

3. *Phase L: Searching for a linear distinguisher on the last segment with Goldreich-Levin.* For each mask Γ_m that survives Phase M, we search for an output mask Γ_o that maximizes

$$q = \left|\mathbb{E}_{K,X}\left[(-1)^{\langle \Gamma_m, X\rangle \oplus \langle \Gamma_o, E_{\ell,K}(X)\rangle}\right]\right|.$$

We define the Boolean oracle $f_K(y) = \langle \Gamma_m, E_{\ell,K}^{-1}(y)\rangle$ on the last segment E_ℓ. A large q indicates that f_K has a heavy Fourier coefficient at frequency Γ_o. The Goldreich-Levin algorithm recovers such coefficients by probing f_K on a small random subspace. For each Γ_m, we run one search on a random key and obtain a candidate Γ_o. Afterwards, we estimate q over several keys to avoid key-dependent artifacts and discard candidates whose average q (across keys) falls below a fixed threshold. Thereby, we obtain a table of tuples $\texttt{LLT}[\Gamma_m, \Gamma_o] = q$.

4. *Ranking Phase.* We assign a score to each tuple $(\Delta_i, \Delta_m, \Gamma_m, \Gamma_o)$ as an estimate of the overall correlation of the corresponding differential-linear distinguisher:

$$\text{Score} \coloneqq |\text{Corr}| = p \cdot |r| \cdot q^2.$$

We retain the top k candidates (e.g., $k = 10$) and discard the rest.

5. *Verification Phase.* We recompute p (and q) forward on U (resp. L) with (Δ_i, Δ_m) (resp. (Γ_m, Γ_o)) using a larger sample size and many keys. We then filter candidates with negligible signal by comparing the observed correlation against the noise level induced by the sample size, and we update their scores accordingly. Finally, we reorder the top list based on the updated scores.

For completeness, and as we feel it is not too widely known, we give a brief description of Goldreich-Levin in Appendix C.

5.3 Results on Threefry−2×32

We applied the segmented search on Threefry−2×32 and obtained practical differential-linear distinguishers for up to 15 rounds of this 64-bit block cipher. Table 4 summarizes our best results. Note that since the rounds of Threefry use different rotation constants, the differential and linear distinguishers depend on the starting round (`offset`) and the chosen segmentation. The first column of Table 4 specifies the number of rounds, the segmentations, as well as the parameters used in Phase L (the Goldreich-Levin algorithm).

Table 4: Specification of differential-linear distinguishers for Threefry.

Configuration		Distinguishers	Parameters
Rounds: $1+9+1=11$	Δ_i:	0x82000000,0x02000000	p: $2^{-1.00}$
Segment: [3–3] + [4–12] + [13–13]	Δ_m:	0x80000000,0x00000000	r: $2^{-4.67}$ (sign +)
Experiments: 2^{16}, Keys: 2^5	Γ_m:	0x00000001,0x00000000	q: $2^{0.00}$
$t=12$, $R=14$, $\tau=3.125\times10^{-2}$	Γ_o:	0x20000001,0x20000000	Cor: $2^{-5.67}$
Rounds: $2+8+2=12$	Δ_i:	0x80050000,0x00040000	p: $2^{-3.99}$
Segment: [0–1] + [2–9] + [10–11]	Δ_m:	0x80000000,0x00000000	r: $2^{-6.56}$ (sign -)
Experiments: 2^{16}, Keys: 2^8	Γ_m:	0x00000000,0x00000040	q: $2^{0.00}$
$t=12$, $R=14$, $\tau=3.1\times10^{-2}$	Γ_o:	0x00000001,0x00000000	Cor: $-2^{-10.54}$
Rounds: $3+8+2=13$	Δ_i:	0x82104020,0x00100020	p: $2^{-7.55}$
Segment: [2–4] + [5–12] + [13–14]	Δ_m:	0x80000000,0x00000000	r: $2^{-5.94}$ (sign -)
Experiments: 2^{16}, Keys: 2^6	Γ_m:	0x00000000,0x00000002	q: $2^{-1.00}$
$t=12$, $R=14$, $\tau=3.125\times10^{-2}$	Γ_o:	0x60000000,0x00000000	Cor: $-2^{-15.49}$
Rounds: $3+8+3=14$	Δ_i:	0x82104020,0x00100020	p: $2^{-7.48}$
Segment: [2–4] + [5–12] + [13–15]	Δ_m:	0x80000000,0x00000000	r: $2^{-5.73}$ (sign -)
Experiments: 2^{23}, Keys: 2^6	Γ_m:	0x00000000,0x04000000	q: $2^{-1.59}$
$t=12$, $R=15$, $\tau=2.762\times10^{-3}$	Γ_o:	0x00808000,0x00008000	Cor: $-2^{-16.38}$
Rounds: $4+8+3=15$	Δ_i:	0x22884004,0x02080100	p: $2^{-16.72}$
Segment: [1–4] + [5–12] + [13–15]	Δ_m:	0x00000100,0x00000000	r: $2^{-7.27}$ (sign -)
Experiments: 2^{20}, Keys: 2^6	Γ_m:	0x00000000,0x00000008	q: $2^{0.00}$
$t=12$, $R=15$, $\tau=2.0\times10^{-3}$	Γ_o:	0x01000001,0x01000000	Cor: $-2^{-23.99}$

The relatively small block size makes it possible to select large middle segments (up to 9 rounds) while keeping the computations in Phase M manageable. All experiments were run on a standard laptop (Apple M4 chip (10 core CPU) in MacBook Pro (Mac16.1)), and the runtime for each configuration ranged from a few minutes to a few hours.

We also investigated whether these distinguishers can turn into key-recovery attacks. Since Threefry−2×32 has only a 64-bit key, there is little room for large guessed-key spaces, and the surrounding modular key additions together with the gap between the distinguisher core and the nearest subkey injections make too many key bits influence the key-recovery step. Consequently, the resulting time complexity is not below exhaustive search, so these distinguishers do not lead to a sub-brute-force key-recovery attack.

6 Polster: A Replacement for Philox and Threefry

In this section, we introduce a replacement for Philox and Threefry that provides cryptographic security, while being well suited for PRNGs on GPU architectures.

Our replacement is based on a reduced-round variant of the Speck family of block ciphers [BSS+13, BSS+15] due to its performance on GPUs [TL25], specifically a modified version of Speck-64-128, which has a 64 bit block and a 128 bit key size. Speck is an Addition-Rotation-XOR based cipher that together with Simon, an And-Rotation-XOR-based cipher, was designed by NSA as lightweight ciphers optimized for hardware and software implementations. Since their introduction, the cryptographic community has extensively analyzed these ciphers, which has increased confidence in their security properties.

In the following, we first explain the Speck block cipher family and describe its encryption algorithm. We then briefly summarize the previous cryptanalysis applied to various versions of Speck. Next, we provide the rationale for selecting the specific Speck variant suitable for our use case. Based on this selection, we present Polster, a reduced-round version of the chosen Speck cipher with a modified key schedule, tailored for high-throughput GPU applications. Finally, we report implementation results and compare the performance of Polster with Philox and Threefry.

Table 5: The parameters for SPECK family of block ciphers.

Block size	$2n$	32	48	48	64	64	96	96	128	128	128
Key size	mn	64	72	96	96	128	96	144	128	192	256
Number of rounds	R	22	22	23	26	27	28	29	32	33	34

6.1 Speck Family of Block Ciphers

SPECK is a family of ARX based block ciphers, designed to provide excellent performance in both hardware and software, with a particular focus on microcontroller environments. The family consists of ten variants, distinguished by their block size and key size parameters. Each cipher operates on a block of two n-bit words (i.e., two branches of n bits), while the key size is composed of m words, with $m \in \{1, 2, 3\}$. Accordingly, a given instance of the cipher is denoted by SPECK-$2n$-mn, where $2n$ specifies the block length in bits and mn indicates the key length in bits. The recommended number of rounds depends on the specific parameters of the variant. All ten versions of the SPECK family are summarized in Table 5.

The round function of SPECK is conceptually similar to the mixing functions used in the THREEFISH block cipher, and therefore also resembles those of THREEFRY. The key differences are that in SPECK all rounds use an identical function, and a round key is added in every round.

The key schedule of SPECK uses another ARX round function to generate the round keys K_i. Let K be the master key of a SPECK-$2n$ block cipher. We represent K as $(L_{m-2}, \ldots, L_0, K_0)$, where each L_i, and K_0 are n-bit key words and $m \in \{2, 3, 4\}$ denotes the number of words in the key. The sequences of round keys K_i and intermediate words L_i are then defined iteratively as follows:

$$L_{m+i+1} = \big(L_i + (L_i \ggg \alpha)\big) \oplus i\,, \quad K_{i+1} = \big(K_i \lll \beta\big) \oplus L_{m+i+1}\,.$$

6.2 Cryptanalysis Results on Speck

The designers of SPECK claimed that, although it is a *lightweight* block cipher, it was intended to provide full security for each block and key size against standard chosen-plaintext (CPA) and chosen-ciphertext (CCA) attacks.

To date, no successful attack on the full-round version of any SPECK variant has been reported. However, the cipher has attracted significant cryptanalytic attention, and numerous studies have been conducted on its reduced-round variants, including differential [ALLW13,BRV14,SHY16,BVC16,DM18,Deh19,QW22,SM23,DCC23,JLM+25], linear(-hull) [BVC16,HW20], rectangle [ALLW13], boomerang [KKS20], integral [EKKT18, HY24], machine-learning-based distinguishability [DNS24, HRC25] cryptanalysis. The most effective published attacks on SPECK in the standard attack models are based on differential cryptanalysis. These attacks typically reach about 70–75% of the total rounds for most variants, but they only achieve marginal improvements over brute force search.

According to the designers, differential attacks were identified as the limiting factor during the design process, i.e., the class of attacks capable of penetrating the largest number of rounds. The final choice of the number of rounds was therefore set to ensure a security margin comparable to that of AES–128, at approximately 30% [BSS+17].

6.3 Specification of Polster

In the following, we explain the rationale behind selecting a specific SPECK variant that best fits our GPU-based PRNG scenario. Building on this choice, we then introduce POLSTER, specifically optimized for high-throughput GPU implementations while maintaining strong

cryptographic properties. The SPECK family of block ciphers includes variants operating on different word sizes, namely $n \in \{16, 24, 32, 48, 64\}$. Among these, the versions with $n = 32$ and $n = 64$ are the most suitable for GPU architectures.

Block Size. Between the two suitable options, $n = 32$ is typically preferable for GPU-based PRNGs. First, most GPU programming models (such as CUDA and OpenCL) expose 32-bit integer arithmetic as the native data type, with 64-bit operations often incurring additional latency or requiring more hardware resources. Second, GPU memory hierarchies are generally optimized for 32-bit words, meaning that state representation, memory transfers, and coalesced access patterns achieve higher efficiency at this granularity. Finally, since PRNGs for large-scale simulations or machine learning workloads often prioritize parallel throughput over individual stream strength, the reduced per-thread state size of $n = 32$ allows more concurrent threads to be scheduled on the GPU, further improving overall performance. Based on this reasoning, we select SPECK-64 as the candidate block cipher for our GPU-based PRNG.

Key Size. Among the two available key sizes, 96-bit and 128-bit, we prefer the 128-bit key for several reasons. First, a larger key provides a higher security margin against brute force attacks, which is especially important when the cipher is used as the core of a high-throughput PRNG where large volumes of data are generated. Second, additional key material allows for more flexibility in the key schedule, improving diffusion across the rounds, and reducing potential structural weaknesses. Finally, the performance overhead of using the 128-bit key schedule compared to the 96-bit one is negligible on modern GPUs, since the per-round operations dominate the runtime and are independent of key length. Thus, SPECK-64-128 strikes a practical balance between security, efficiency, and suitability for GPU implementations.

Number of Rounds. The strongest published attack (with respect to the highest number of rounds) on SPECK-64-128 is a differential attack targeting 20 rounds, with a complexity of $2^{125.56}$ encryptions and requiring $2^{61.56}$ chosen plaintexts. This attack, together with many of the other attacks on SPECK, exploits a recurring technical convenience: In numerous cryptanalyses, it is possible to prepend one round to the distinguisher without introducing additional key guesses. This is because adding the round key is equivalent to applying it to the state after all other round operations have been performed. Concretely, for differential cryptanalysis, if a differential distinguisher begins at the second round, we may choose plaintext pairs so that after the first round the state difference equals the input difference of the distinguisher; this allows the distinguisher to be applied to an additional round without guessing any key material.

A similar technique is commonly used in the key-recovery extension of attacks: the right branch at the input of the last round can often be recovered without guessing additional key material. Both of these issues can be mitigated by a simple modification of adding pre-whitening and post-whitening key additions before the first round and after the last round, respectively.

Moreover, the most effective published attacks that reach (or approach) the highest round counts incur very large time and/or data complexity. Consequently, taking into account the whitening countermeasure and the high complexity of the best known attacks, we believe that 20 rounds provide a sufficient security margin for our use case.

Key Schedule. Each round function of SPECK consists of two `ROT`, two `XOR`, and one `ADD` operation. Similarly, each update of the key schedule employs exactly the same set and number of operations. This implies that, in a GPU implementation of SPECK, roughly half of the computational cost is incurred by the key schedule if the round keys were

Algorithm 1: POLSTER Encryption.

Input: $P = (X, Y)$ and $K = (K_0, K_1, K_2, K_3)$

$X \leftarrow X + K_2$;
$Y \leftarrow Y + K_3$;
$K_2 \leftarrow K_2 + C_2$;
$K_3 \leftarrow K_3 + C_3$;
for $r = 0$ *increasing to* $r = 19$ **do**
  $i \leftarrow r \bmod 4$;
  $X \leftarrow \big((X \ggg 8) + Y\big) \oplus K_i$;
  $Y \leftarrow \big(Y \lll 3\big) \oplus X$;
  $K_i \leftarrow K_i + C_i$;
$X \leftarrow X + K_0$;
$Y \leftarrow Y + K_1$;
return $C = (X, Y)$;

not pre-computed at the CPU side and transferred to the GPU memory. Consequently, adopting an alternative key schedule that requires fewer operations per update, while maintaining comparable security, can lead to a significant performance improvement.

We propose using a key schedule similar to the one in PHILOX. Let the user key be $K = (K_0, K_1, K_2, K_3)$, where each K_i is a 32-bit word. For the r-th round, the round key is taken as $K_{r \bmod 4}$, and then updated as

$$K_{r \bmod 4} \leftarrow K_{r \bmod 4} + C_{r \bmod 4}\,,$$

where the addition is modulo 2^{32} and the constants are

$$C_0 = \texttt{0xBB67AE85}\,,\ C_1 = \texttt{0x84CAA73B}\,,\ C_2 = \texttt{0x9E3779B9}\,,\ C_3 = \texttt{0x7F4A7C15}\,,$$

which are derived from the constants used in PHILOX. With this approach, the key schedule requires only a single `ADD` per round, which is significantly lower than the cost of the original SPECK key schedule.

Specification of Polster. After introducing the modified key schedule, we present POLSTER, detailed in Algorithm 1. It follows the reduced-round structure of SPECK-64-128 variant while incorporating our lightweight, PHILOX-like key schedule. Note that while round keys are combined with the state using `XOR`, the whitening keys are applied via modular addition. We provide some test vectors for POLSTER that are listed in Table 7.

6.4 Implementations Results

We used the benchmark[5] provided by [SMDS11] on different GPUs to obtain CTR mode of operation results for PHILOX – 4×32 – 10 and THREEFRY – 4×32 – 20. We observed that on older GPUs, THREEFRY – 4×32 – 20 outperforms PHILOX – 4×32 – 10. However, on modern GPUs it's vice versa. This suggests that architectural changes in the future might affect the performance of random number generators and having multiple alternatives might be beneficial.

We provide CTR mode of encryption results of SPECK, POLSTER, PHILOX, and THREEFRY in Table 6. Although PHILOX – 4×32 – 10 and THREEFRY – 2×32 – 20 provide fast results, we showed in Appendix A that their 64-bit key can be practically captured via brute force attacks.

[5] https://github.com/DEShawResearch/random123

Table 6: Performance comparison of Polster with Philox, Threefry, and Speck. Note that our results for Philox are from the brute force implementation, and the reported results are in a unit of [GB/s].

Primitive	MX250	GTX860M	RTX2070 Super	RTX4090	Ref.
Philox – 4×32 – 10	50.2	52.8	1800.0	8190.0	[SMDS11]
Philox – 4×32 – 10	53.9	52.5	1397.9	5924.1	[Ours]
Threefry – 2×32 – 20	57.1	48.9	675.0	3057.5	[Ours]
Speck-64-128	33.4	29.2	302.6	1387.1	[Ours]
Polster-64-128	38.7	35.1	327.8	1537.7	[Ours]

7 Conclusion and Future Work

In this paper, we present the first (and long-overdue) review of the cryptographic strength provided by the most commonly used PRNGs for AI applications. As it turns out, given our brute force attacks on GPUs and the cryptographic shortcomings of the designs, the most commonly used PRNGs do not provide the level of protection required by modern AI applications. Therefore, we propose Polster as a secure and well-established alternative.

While our focus was on providing a well-understood alternative rather than introducing a new GPU-optimized design, future work might explore the development of such a GPU-optimized, cryptographically strong primitive.

References

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016. URL: http://arxiv.org/abs/1603.04467, arXiv:1603.04467.

[ALLW13] Farzaneh Abed, Eik List, Stefan Lucks, and Jakob Wenzel. Cryptanalysis of the speck family of block ciphers. *IACR Cryptol. ePrint Arch.*, page 568, 2013. URL: http://eprint.iacr.org/2013/568.

[Bai25a] Baidu. Paddlepaddle: Parallel distributed deep learning platform. https://github.com/PaddlePaddle/Paddle, 2025. Accessed: 2025-09-17.

[Bai25b] Baidu Inc. Paddlepaddle, 2025. URL: https://github.com/PaddlePaddle/Paddle.

[BCJW02] Nikita Borisov, Monica Chew, Robert Johnson, and David A. Wagner. Multiplicative differentials. In Joan Daemen and Vincent Rijmen, editors, *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers*, volume 2365 of *LNCS*, pages 17–33. Springer, 2002. doi:10.1007/3-540-45661-9_2.

[BDKW19] Achiya Bar-On, Orr Dunkelman, Nathan Keller, and Ariel Weizman. DLCT: A new tool for differential-linear cryptanalysis. In *EUROCRYPT 2019*, volume

11476 of *LNCS*, pages 313–342. Springer, 2019. doi:10.1007/978-3-030-17653-2_11.

[BFH+18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL: http://github.com/jax-ml/jax.

[BGG+23] Emanuele Bellini, David Gérault, Juan Grados, Rusydi H. Makarim, and Thomas Peyrin. Fully automated differential-linear attacks against ARX ciphers. In Mike Rosulek, editor, *CT-RSA 2023*, volume 13871 of *LNCS*, pages 252–276. Springer, 2023. doi:10.1007/978-3-031-30872-7_10.

[BLN17] Céline Blondeau, Gregor Leander, and Kaisa Nyberg. Differential-linear cryptanalysis revisited. *J. Cryptol.*, 30(3):859–888, 2017. doi:10.1007/s00145-016-9237-5.

[BRV14] Alex Biryukov, Arnab Roy, and Vesselin Velichkov. Differential analysis of block ciphers SIMON and SPECK. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*, pages 546–570. Springer, 2014. doi:10.1007/978-3-662-46706-0_28.

[BSS+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. *IACR Cryptol. ePrint Arch.*, page 404, 2013. URL: http://eprint.iacr.org/2013/404.

[BSS+15] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 175:1–175:6. ACM, 2015. doi:10.1145/2744769.2747946.

[BSS+17] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. Notes on the design and analysis of SIMON and SPECK. *IACR Cryptol. ePrint Arch.*, page 560, 2017. URL: http://eprint.iacr.org/2017/560.

[BVC16] Alex Biryukov, Vesselin Velichkov, and Yann Le Corre. Automatic search for the best trails in ARX: application to block cipher speck. In Thomas Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 289–310. Springer, 2016. doi:10.1007/978-3-662-52993-5_15.

[CLL+15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. URL: http://arxiv.org/abs/1512.01274, arXiv:1512.01274.

[DCC23] Haoran Deng, Xianghui Cao, and Yu Cheng. Attention in differential cryptanalysis on lightweight block cipher SPECK. In *20th Annual International Conference on Privacy, Security and Trust, PST 2023, Copenhagen, Denmark, August 21-23, 2023*, pages 1–9. IEEE, 2023. doi:10.1109/PST58708.2023.10320201.

[Deh19] Seyed Mojtaba Dehnavi. Further observations on SIMON and SPECK block cipher families. *Cryptogr.*, 3(1):1, 2019. URL: https://doi.org/10.3390/cryptography3010001, doi:10.3390/CRYPTOGRAPHY3010001.

[DFE+25] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention, 2025. URL: https://github.com/Dao-AILab/flash-attention.

[DM18] Ashutosh Dhar Dwivedi and Pawel Morawiecki. Differential cryptanalysis in ARX ciphers, application to SPECK. *IACR Cryptol. ePrint Arch.*, page 899, 2018. URL: https://eprint.iacr.org/2018/899.

[DNS24] Jimmy Dani, Kalyan Nakka, and Nitesh Saxena. Breaking indistinguishability with transfer learning: A first look at SPECK32/64 lightweight block ciphers. *IACR Cryptol. ePrint Arch.*, page 852, 2024. URL: https://eprint.iacr.org/2024/852.

[DR14] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014. doi:10.1561/0400000042.

[DSA24] Pranav Dahiya, Ilia Shumailov, and Ross Anderson. Machine learning needs better randomness standards: Randomised smoothing and prng-based attacks. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/dahiya.

[EKKT18] Zahra Eskandari, Andreas Brasen Kidmose, Stefan Kölbl, and Tyge Tiessen. Finding integral distinguishers with ease. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, volume 11349 of *Lecture Notes in Computer Science*, pages 115–138. Springer, 2018. doi:10.1007/978-3-030-10970-7_6.

[Fac25] Hugging Face. Transformers documentation. https://huggingface.co/docs/transformers/en/index, 2025. Accessed: 2025-09-17.

[Fur01] Vladimir Furman. Differential cryptanalysis of nimbus. In Mitsuru Matsui, editor, *Fast Software Encryption, 8th International Workshop, FSE 2001 Yokohama, Japan, April 2-4, 2001, Revised Papers*, volume 2355 of *LNCS*, pages 187–195. Springer, 2001. doi:10.1007/3-540-45473-X_16.

[GKN+26] Nir Goren, Oren Katzir, Abhinav Nakarmi, Eyal Ronen, Mahmood Sharif, and Or Patashnik. Noiseprints: Distortion-free watermarks for authorship in private diffusion models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL: https://openreview.net/forum?id=gwHCXvPDBd.

[Goo25a] Google Brain Team. Tensorflow, 2025. URL: https://www.tensorflow.org/.

[Goo25b] Google DeepMind. Jax, 2025. URL: https://github.com/jax-ml/jax.

[GWH+25] Xinxin Gong, Qingju Wang, Yonglin Hao, Lin Jiao, and Xichao Hu. Persistence of hourglass(-like) structure: Improved differential-linear distinguishers for several ARX ciphers. Cryptology ePrint Archive, Paper 2025/1667, 2025. https://ia.cr/2025/1667.

[HDE24] Hosein Hadipour, Patrick Derbez, and Maria Eichlseder. Revisiting differential-linear attacks via a boomerang perspective with application to AES, Ascon, CLEFIA, SKINNY, PRESENT, KNOT, TWINE, WARP, LBlock, Simeck, and SERPENT. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024*, volume 14923 of *LNCS*, pages 38–72. Springer, 2024. doi:10.1007/978-3-031-68385-5_2.

[HJA20] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/4c5bcfec8584af0d967f1ab10179ca4b-Abstract.html.

[HRC25] Zezhou Hou, Jiongjiong Ren, and Shaozhen Chen. Further observations on the security of speck32-like ciphers using machine learning. *PeerJ Comput. Sci.*, 11:e3015, 2025. URL: https://doi.org/10.7717/peerj-cs.3015, doi:10.7717/PEERJ-CS.3015.

[HW20] Mingjiang Huang and Liming Wang. Automatic search for the linear (hull) characteristics of ARX ciphers: Applied to speck, sparx, chaskey, and CHAM-64. *Secur. Commun. Networks*, 2020:4898612:1–4898612:14, 2020. doi:10.1155/2020/4898612.

[HY24] Kai Hu and Trevor Yap. Perfect monomial prediction for modular addition. *IACR Trans. Symmetric Cryptol.*, 2024(3):177–199, 2024. URL: https://doi.org/10.46586/tosc.v2024.i3.177-199, doi:10.46586/TOSC.V2024.I3.177-199.

[JGP17] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=rkE3y85ee.

[JLM+25] Xue Jiang, Min Li, Kaiyrbek Makulov, Valerii Lakhno, and Andrii Sahun. Enhanced neural differential distinguisher for speck32/64 using attention mechanisms and multi ciphertext inputs. *Informatica (Slovenia)*, 49(19), 2025. URL: https://doi.org/10.31449/inf.v49i19.7889, doi:10.31449/INF.V49I19.7889.

[KFP+25] Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean-Bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Róbert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen,

Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucinska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju-yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle K. Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clément Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry (Dima) Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. Gemma 3 technical report. *CoRR*, abs/2503.19786, 2025. URL: https://doi.org/10.48550/arXiv.2503.19786, arXiv:2503.19786, doi:10.48550/ARXIV.2503.19786.

[KKS20] Dongyeong Kim, Dawoon Kwon, and Junghwan Song. Efficient computation of boomerang connection probability for arx-based block ciphers with application to SPECK and LEA. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 103-A(4):677–685, 2020. URL: https://doi.org/10.1587/transfun.2019EAP1083, doi:10.1587/TRANSFUN.2019EAP1083.

[LeC98] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[LS07] Pierre L'Ecuyer and Richard J. Simard. Testu01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4):22:1–22:40, 2007. doi:10.1145/1268776.1268777.

[LZQ+] Chao Liu, Jing Zhang, Letao Qin, Qianfeng Zhang, Liang Huang, Shaojie Wang, Anthony Chang, Chunyu Lai, Illia Silin, Adam Osewski, Poyen Chen, Rosty Geyyer, Hanwen Chen, Tejash Shah, Xiaoyan Zhou, and Jianfeng Yan. Composable kernel. License file pinned at commit 7fc3ed761aa35709d87c8fbbe41dd368648b3541. URL: https://github.com/ROCm/composable_kernel.

[Met25] Meta Platforms, Inc. Opacus, 2025. URL: https://opacus.ai/.

[Mic23] Microsoft Security Response Center. Microsoft vulnerability severity and content classifications for AI systems, 2023. Accessed: 2026-02-10. URL: https://www.microsoft.com/en-us/msrc/aibugbar.

[MSL+25] Felix Mächtle, Jan-Niclas Serr, Nils Loose, Jonas Sander, and Thomas Eisenbarth. OCEAN: open-world contrastive authorship identification. In Marc Fischlin and Veelasha Moonsamy, editors, *Applied Cryptography and Network Security - 23rd International Conference, ACNS 2025, Munich, Germany, June 23-26, 2025, Proceedings, Part II*, volume 15826 of *LNCS*, pages 459–486. Springer, 2025. doi:10.1007/978-3-031-95764-2_18.

[MSS+26] Felix Mächtle, Ashwath Shetty, Jonas Sander, Nils Loose, Sören Pirk, and Thomas Eisenbarth. Prompt pirates need a map: Stealing seeds helps stealing prompts. In Manindra Agrawal, Ian Molloy, Vinayaka Pandit, Debdeep Mukhopadhyay, and Kenny Paterson, editors, *Proceedings of the ACM Asia Conference on Computer and Communications Security, ASIA CCS 2026, Bangalore, India, June 1-5, 2026*, pages 422–438. ACM, 2026. doi:10.1145/3779208.3807483.

[NVE+25] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[NVI25a] NVIDIA. Apex: Tools for easy mixed precision and distributed training in pytorch. GitHub repository, 2025. Version 25.09 (commit 4bdecd0); License: BSD-3-Clause. URL: https://github.com/NVIDIA/apex.

[NVI25b] NVIDIA Corporation & Affiliates. Achieving high performance, 2025. URL: https://docs.nvidia.com/cuda/curanddx/get_started/performance.html.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[PyT25] PyTorch Foundation. Pytorch, 2025. URL: https://pytorch.org/.

[QW22] Haiwen Qin and Baofeng Wu. Towards non-independence of modular additions in searching differential trails of ARX ciphers: new automatic methods with application to SPECK and chaskey. *CoRR*, abs/2203.09741, 2022. URL: https://doi.org/10.48550/arXiv.2203.09741, arXiv:2203.09741, doi:10.48550/ARXIV.2203.09741.

[SHY16] Ling Song, Zhangjie Huang, and Qianqian Yang. Automatic differential analysis of ARX block ciphers with application to SPECK and LEA. In Joseph K. Liu and Ron Steinfeld, editors, *Information Security and Privacy - 21st*

Australasian Conference, ACISP 2016, Melbourne, VIC, Australia, July 4-6, 2016, Proceedings, Part II, volume 9723 of *Lecture Notes in Computer Science*, pages 379–394. Springer, 2016. doi:10.1007/978-3-319-40367-0_24.

[SM23] Ayan Sajwan and Girish Mishra. Comparative analysis of resnet and densenet for differential cryptanalysis of SPECK 32/64 lightweight block cipher. *IACR Cryptol. ePrint Arch.*, page 1290, 2023. URL: https://eprint.iacr.org/2023/1290.

[SMDS11] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In Scott Lathrop, Jim Costa, and William Kramer, editors, *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, pages 16:1–16:12. ACM, 2011. doi:10.1145/2063384.2063405.

[Sta24] Stability AI. Introducing stable diffusion 3.5. https://stability.ai/news/introducing-stable-diffusion-3-5, 2024. Accessed: 2025-09-17.

[Tez21] Cihangir Tezcan. Optimization of advanced encryption standard on graphics processing units. *IEEE Access*, 9:67315–67326, 2021. doi:10.1109/ACCESS.2021.3077551.

[The23] The Apache Software Foundation. Apache mxnet, 2023. URL: https://github.com/apache/mxnet.

[TL25] Cihangir Tezcan and Gregor Leander. GPU assisted brute force cryptanalysis of GPRS, GSM, RFID, and TETRA. *IACR Trans. Symmetric Cryptol.*, 2025(1):309–327, 2025. URL: https://doi.org/10.46586/tosc.v2025.i1.309-327, doi:10.46586/TOSC.V2025.I1.309-327.

[TLH25] Cihangir Tezcan, Gregor Leander, and Hosein Hadipour. Cryptanalysis: Theory versus practice: Correcting cryptanalysis results on Ascon, ChaCha, and Serpent using GPUs. *IACR Transactions on Symmetric Cryptology*, 2025(3):729–754, Sep. 2025. URL: https://tosc.iacr.org/index.php/ToSC/article/view/12484, doi:10.46586/tosc.v2025.i3.729-754.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[WAW+25] Ang Wang, Baole Ai, Bin Wen, Chaojie Mao, Chen-Wei Xie, Di Chen, Feiwu Yu, Haiming Zhao, Jianxiao Yang, Jianyuan Zeng, Jiayu Wang, Jingfeng Zhang, Jingren Zhou, Jinkai Wang, Jixuan Chen, Kai Zhu, Kang Zhao, Keyu Yan, Lianghua Huang, Xiaofeng Meng, Ningyi Zhang, Pandeng Li, Pingyu Wu, Ruihang Chu, Ruili Feng, Shiwei Zhang, Siyang Sun, Tao Fang, Tianxing Wang, Tianyi Gui, Tingyu Weng, Tong Shen, Wei Lin, Wei Wang, Wei Wang, Wenmeng Zhou, Wente Wang, Wenting Shen, Wenyuan Yu, Xianzhong Shi, Xiaoming Huang, Xin Xu, Yan Kou, Yangyu Lv, Yifei Li, Yijing Liu, Yiming Wang, Yingya Zhang, Yitong Huang, Yong Li, You Wu, Yu Liu, Yulin Pan, Yun Zheng, Yuntao Hong, Yupeng Shi, Yutong Feng, Zeyinzi Jiang, Zhen Han, Zhi-Fan Wu, and Ziyu Liu. Wan: Open and advanced large-scale video generative models. *CoRR*, abs/2503.20314, 2025. URL:

https://doi.org/10.48550/arXiv.2503.20314, arXiv:2503.20314, doi:10.48550/ARXIV.2503.20314.

[YSS+21] Ashkan Yousefpour, Igor Shilov, Alexandre Sablayrolles, Davide Testuggine, Karthik Prasad, Mani Malek, John Nguyen, Sayan Ghosh, Akash Bharadwaj, Jessica Zhao, Graham Cormode, and Ilya Mironov. Opacus: User-friendly differential privacy library in pytorch. *CoRR*, abs/2109.12298, 2021. URL: https://arxiv.org/abs/2109.12298, arXiv:2109.12298.

A Brute Force on GPUs

In this section, we discuss the cost of an exhaustive search for both Philox and Threefry. We added our GPU codes as supplementary material to this submission and will make them publicly available after the paper is published.

A.1 Preliminaries on GPUs

Modern GPUs have tens of thousands of cores which provide significant speedups for parallel applications when compared with CPUs. This speedup is more significant when the threads do not communicate with each other since data transfers can be the bottleneck. Brute force attacks on block ciphers and counter (CTR) mode of operation for block ciphers are good examples for applications where threads do not need to communicate with each other.

In a brute force attack, the key space can be divided into disjoint subsets where each thread searches the key in its subset for a captured plaintext-ciphertext pair. In the CTR mode of operation, each thread uses a different counter for a fixed key and this time the counter space can be divided into disjoint subsets. Both of these approaches can be used for random number generation. They might look very similar with the only difference in which one of them increments the key and the other one increments the counter. However, the CTR mode can perform better for random number generation because the key schedule part can be eliminated by performing it only once at the CPU side for the fixed key and transferring the round keys to GPU memory, if the context allows such communication.

The set of hardware and software features of a CUDA device is defined by the device compute capability and GPUs are backward compatible with respect to it. For example, a GPU kernel compiled for compute capability 5.2 can be run on a GPU that has a compute capability of 8.9. The performance of a GPU kernel can change depending on the architecture of the GPU, compute capability, the number of threads used in each block, and the used CUDA SDK. Thus, in this work we compiled our codes for every supported compute capability, many different combinations of thread and block numbers, and many different CUDA SDKs. We obtained the best performance when we used 1024 threads and compute capability 5.2 provided the best results for Speck. For best occupancy, we chose number of threads and number of blocks as 1024. Thus, each thread performs multiple encryptions when more than 2^{20} keys are tried.

Although Philox and Threefry have versions with 32-bit or 64-bit words, modern GPU registers are 32-bit and 64-bit integer operations require execution of two 32-bit integer instructions. Thus, doubling the words size does not generally double the performance. For example, when we used the designers' publicly available source codes, we achieved 67.0, 68.6, and 74.6 GB/s on our GTX860M GPU for Threefry – 4×32 – 20, Threefry – 2×64 – 20, and Threefry – 4×64 – 20, respectively.

A.2 Philox and Threefry on GPUs

Philox uses multiplication operation which expands the word size and doubling the word size for multiplication introduces a significant slowdown on GPUs. For this reason, `__umulhi()` CUDA instruction is used to calculate the most significant 32 bits of the product of the two 32-bit unsigned integers. The least significant 32 bits of the product is calculated simply by multiplying the two 32-bit unsigned integers. This approach provides better performance than multiplying two 64-bit words.

Similarly, we used `__umul64hi()` CUDA instruction for the 64-bit version of Philox to calculate the most significant 64 bits of the product of the two 64-bit unsigned integers. This way we are not required to work with 128-bit integers.

We observed that Philox runs significantly faster on newer architectures suggesting that `__umulhi()` and `__umul64hi()` integer intrinsics require less clock cycles on newer CUDA GPUs.

Philox has two variants with 64-bit keys. We run our experiments to try 2^{45} keys and we observed that our optimizations achieve $2^{38.53}$ and $2^{37.85}$ keys per second for Philox – 4×32 – 10 and Philox – 2×64 – 10, respectively, on a single RTX 4090 GPU. Thus, an RTX 4090 GPU can capture the key for Philox – 4×32 – 10 in 537 days or similarly, 537 RTX 4090 GPUs can capture the key in a day. This number increases to 1038 for Philox – 2×64 – 10 since using 64-bit integers slows it down on GPUs.

Similarly, for Threefry, we run our experiments to try 2^{45} keys and we observed that our optimizations achieve $2^{38.58}$ keys per second for Threefry – 2×32 – 20 on a single RTX 4090 GPU. Thus, an RTX 4090 GPU can capture the key in 521 days or similarly, 521 RTX 4090 GPUs can capture the key in a day.

We note that a single RTX 4090 costs around $2200 at the time of writing. Moreover, many cloud services charge less than $0.6 per hour for an RTX 4090, reducing the one-time cost of these attacks to below $8,323, $14,947, and $7,502 for Philox – 4×32 – 10, Philox – 2×64 – 10, and Threefry – 2×32 – 20, respectively.

Note that the brute force performance of RTX 4090 is compared for the ciphers AES, DES, Kasumi, Klein, Present, Speck, and TEA3 in [TL25] and for the ciphers Ascon, ChaCha, and Serpent in [TLH25]. The best performance was achieved for Speck-64-96 with $2^{36.72}$ keys per second. Achieving almost four times better speed for Philox and Threefry shows the effect of simplified round functions and small number of rounds of these algorithms.

Finally, in our brute force attacks we assumed that the attacker has access to a single input and output and encrypt the input with every possible key to check if the captured output is observed. In an environment where capturing the input is not possible, the brute force can be applied when two consecutive outputs are captured. In that case the attacker decrypts both outputs to see if the obtained inputs are consecutive. This way the attacker captures both the secret key and the counter. Moreover, this second approach only doubles the attack duration.

B Additional Distinguishers on Philox

We start by presenting two distinguishers for Philox – 2×W that isolate the main mechanism behind our attacks. We view these theorems as a compact blueprint: the Philox – 2×W structure makes the arguments particularly transparent, and the same reasoning can be lifted to obtain Distinguishers 5 and 6 for Philox – 4×W, the practically more interesting instance.

The first distinguisher is based on multiplicative differentials.

Distinguisher 3. *Let* $P = (L^{(0)}, R^{(0)})$ *and* $\tilde{P} = (\tilde{L}^{(0)}, \tilde{R}^{(0)})$ *be two plaintexts for* PHILOX–2×W*, such that* $L^{(0)}$ *is odd (i.e.,* $L_0^{(0)} = 1$*), and*

$$\begin{cases} \tilde{L}^{(0)} = L^{(0)} \oplus (2^{\mathrm{W}} - 2)\,, \\ \tilde{R}^{(0)} = R^{(0)} \oplus \mathrm{mulhi}_M(L^{(0)}) \oplus \left(M - 1 - \mathrm{mulhi}_M(L^{(0)})\right)\,. \end{cases}$$

Then, after four rounds the corresponding states $(L^{(4)}, R^{(4)})$ *and* $(\tilde{L}^{(4)}, \tilde{R}^{(4)})$ *satisfy the following relation:*

$$\Pr\left[\, R_0^{(4)} = \tilde{R}_0^{(4)} \,\right] = \frac{M-1}{2^{\mathrm{W}}} \approx 0.8\,,$$

where the probability is taken over all values for the round keys and the remaining $2\mathrm{W} - 1$ *bits of plaintext* P*. Besides, the corresponding states after five rounds* $(L^{(5)}, R^{(5)})$ *and* $(\tilde{L}^{(5)}, \tilde{R}^{(5)})$ *satisfy the following relation:*

$$\Pr\left[\, \left(F_M(R^{(5)}) \oplus L^{(5)}\right)_0 = \left(F_M(\tilde{R}^{(5)}) \oplus \tilde{L}^{(5)}\right)_0 \,\right] = \frac{M-1}{2^{\mathrm{W}}} \approx 0.8\,.$$

Proof. Since $\tilde{L}^{(0)} = L^{(0)} \oplus (2^{\mathrm{W}} - 2)$, and both $L^{(0)}$ and $\tilde{L}^{(0)}$ are odd, we have $\tilde{L}^{(0)} = -L^{(0)}$. Therefore,

$$\begin{aligned} \tilde{R}^{(1)} &= \mathrm{mullo}_M(\tilde{L}^{(0)}) = \mathrm{mullo}_M(-L^{(0)}) = -\,\mathrm{mullo}_M(L^{(0)}) = -R^{(1)} \\ &= R^{(1)} \oplus (2^{\mathrm{W}} - 2)\,, \end{aligned}$$

with both $R^{(1)}$ and $\tilde{R}^{(1)}$ being odd. Besides,

$$\begin{aligned} \tilde{L}^{(1)} &= \mathrm{mulhi}_M(\tilde{L}^{(0)}) \oplus \tilde{R}^{(0)} \oplus K^{(0)} \\ &= \mathrm{mulhi}_M(-L^{(0)}) \oplus R^{(0)} \oplus \mathrm{mulhi}_M(L^{(0)}) \oplus \left(M - 1 - \mathrm{mulhi}_M(L^{(0)})\right) \oplus K^{(0)} \\ &= R^{(0)} \oplus \mathrm{mulhi}_M(L^{(0)}) \oplus K^{(0)} = L^{(1)}\,. \end{aligned}$$

Thus, after the second round we obtain

$$\tilde{R}^{(2)} = R^{(2)}\,, \qquad \tilde{L}^{(2)} = L^{(2)} \oplus (2^{\mathrm{W}} - 2)\,.$$

Note that $2^{\mathrm{W}} - 2$ has LSB equal to 0. Hence, $\tilde{L}^{(2)}$ and $L^{(2)}$ always have the same parity. Moreover,

$$L^{(2)} \oplus (2^{\mathrm{W}} - 2) = \begin{cases} -L^{(2)} \pmod{2^{\mathrm{W}}}, & \text{if } L^{(2)} \text{ is odd}, \\ (2^{\mathrm{W}} - 2) - L^{(2)}, & \text{if } L^{(2)} \text{ is even}, \end{cases}$$

and therefore the relations $\tilde{L}^{(2)} \equiv -L^{(2)} \pmod{2^{\mathrm{W}}}$ and $\tilde{L}^{(2)} + L^{(2)} = 2^{\mathrm{W}} - 2$ occur according to whether $L^{(2)}$ is odd or even. Furthermore, we have

$$\begin{aligned} \tilde{L}^{(3)} &= \mathrm{mulhi}_M(\tilde{L}^{(2)}) \oplus \tilde{R}^{(2)} \oplus K^{(2)} \\ &= \mathrm{mulhi}_M(L^{(2)} \oplus (2^{\mathrm{W}} - 2)) \oplus R^{(2)} \oplus K^{(2)} \\ &= \mathrm{mulhi}_M(L^{(2)} \oplus (2^{\mathrm{W}} - 2)) \oplus \mathrm{mulhi}_M(L^{(2)}) \oplus L^{(3)}\,. \end{aligned}$$

Since M is odd, for any X, we have $(\mathrm{mullo}_M(X))_0 = X_0$. Therefore,

$$\begin{aligned} &\Pr\left[R_0^{(4)} = \tilde{R}_0^{(4)}\right] = \Pr\left[\left(\mathrm{mullo}_M(L^{(3)})\right)_0 = \left(\mathrm{mullo}_M(\tilde{L}^{(3)})\right)_0\right] = \\ &\Pr\left[L_0^{(3)} = \tilde{L}_0^{(3)}\right] = \Pr\left[\left(\mathrm{mulhi}_M\left(L^{(2)} \oplus (2^{\mathrm{W}} - 2)\right)\right)_0 = \left(\mathrm{mulhi}_M(L^{(2)})\right)_0\right]\,. \end{aligned}$$

Next, we analyze the probability that the LSB of $\mathrm{mulhi}_M(L^{(2)})$ and the LSB of $\mathrm{mulhi}_M(L^{(2)} \oplus (2^{\mathrm{W}} - 2))$ are equal. To this end, we distinguish two cases depending on

the parity of $L^{(2)}$: either $L^{(2)}$ is odd, or it is even. Note that each of these cases occurs with equal probability.

In the case of $L^{(2)}$ being odd, we have

$$\begin{aligned}\Big(\operatorname{mulhi}_M\left(L^{(2)} \oplus (2^{\mathrm{W}}-2)\right)\Big)_0 &= \left(\operatorname{mulhi}_M(-L^{(2)})\right)_0 \quad = \left(M-1-\operatorname{mulhi}_M(L^{(2)})\right)_0 \\ &= \left(\operatorname{mulhi}_M(L^{(2)})\right)_0 .\end{aligned}$$

Hence, for an odd $L^{(2)}$, we always have

$$\Big(\operatorname{mulhi}_M\left(L^{(2)} \oplus (2^{\mathrm{W}}-2)\right)\Big)_0 = \left(\operatorname{mulhi}_M(L^{(2)})\right)_0 .$$

For the case where $L^{(2)}$ is even, the probability

$$p := \Pr_{X\colon \text{even}}\left[\Big(\operatorname{mulhi}_M\left(X \oplus (2^{\mathrm{W}}-2)\right)\Big)_0 = \left(\operatorname{mulhi}_M(X)\right)_0\right]$$

depends on both the word size W and the choice of multiplier M. In the case of PHILOX, all recommended multipliers satisfy that both their MSB and LSB are equal to 1. Under this condition, the above probability equals

$$p = \frac{M - 2^{\mathrm{W}-1} - 1}{2^{\mathrm{W}-1}} .$$

Combining this with the case where $L^{(2)}$ is odd, we obtain

$$\Pr\left[R_0^{(4)} = \tilde{R}_0^{(4)}\right] = \frac{1}{2}\times 1 + \frac{1}{2}\times\frac{M-2^{\mathrm{W}-1}-1}{2^{\mathrm{W}-1}} = \frac{M-1}{2^{\mathrm{W}}} .$$

The above 4-round differential-linear distinguisher can be extended by one round, by using the fact that $\operatorname{mullo}_M(\cdot)$ is a bijection. From the round update, we have

$$\begin{aligned}R^{(4)} &= \operatorname{mulhi}_M(L^{(4)}) \oplus L^{(5)} \oplus K^{(4)} \\ &= \operatorname{mulhi}_M\left(\operatorname{mullo}_M^{-1}(R^{(5)})\right) \oplus L^{(5)} \oplus K^{(4)} .\end{aligned}$$

Using the definition of F_M, the relation of the 4-round distinguisher can be rewritten as

$$\Pr\left[R_0^{(4)} = \tilde{R}_0^{(4)}\right] = \Pr\left[\left(F_M(R^{(5)}) \oplus L^{(5)}\right)_0 = \left(F_M(\tilde{R}^{(5)}) \oplus \tilde{L}^{(5)}\right)_0\right] ,$$

which naturally yields a 5-round distinguisher on PHILOX–2×W. □ □

Next, we present an alternative distinguisher for the same number of rounds, albeit with a lower bias. The primary significance of this variant lies in its underlying adversarial model. This configuration relaxes the data collection requirements and ultimately enables ciphertext-only attacks.

Distinguisher 4. *Let $P = (L^{(0)}, R^{(0)})$ and $\tilde{P} = (\tilde{L}^{(0)}, \tilde{R}^{(0)})$ be two plaintexts for PHILOX–2×W, such that $\tilde{R}^{(0)} = R^{(0)} \oplus 1$ and $\tilde{L}^{(0)} = L^{(0)}$. That is, the two plaintexts differ only in the LSB of the right word. Then, after four rounds the corresponding states $(L^{(4)}, R^{(4)})$ and $(\tilde{L}^{(4)}, \tilde{R}^{(4)})$ satisfy the following relation:*

$$\Pr\left[R_0^{(4)} = \tilde{R}_0^{(4)}\right] \approx 0.56 ,$$

where the probability is taken over all values for the round keys and the remaining $2\mathrm{W}-1$ bits of plaintext P. Besides, the corresponding states after five rounds $(L^{(5)}, R^{(5)})$ and $(\tilde{L}^{(5)}, \tilde{R}^{(5)})$ satisfy the following relation:

$$\Pr\left[\left(F_M(R^{(5)}) \oplus L^{(5)}\right)_0 = \left(F_M(\tilde{R}^{(5)}) \oplus \tilde{L}^{(5)}\right)_0\right] \approx 0.56 .$$

Proof. Since $\tilde{R}^{(0)} = R^{(0)} \oplus 1$ and $\tilde{L}^{(0)} = L^{(0)}$, we have $\tilde{R}^{(1)} = R^{(1)}$ and $\tilde{L}^{(1)} = L^{(1)} \oplus 1$. Thus, after the first round, the pair differs only in the LSB of the *left* word. Using the fact that M is odd, for any X, we have $(\mathrm{mullo}_M(X))_0 = X_0$, we obtain

$$\tilde{R}_0^{(2)} \oplus R_0^{(2)} = \left(\mathrm{mullo}_M(\tilde{L}^{(1)}) \oplus \mathrm{mullo}_M(L^{(1)})\right)_0 = \tilde{L}_0^{(1)} \oplus L_0^{(1)} = 1\,.$$

At the same time, we have

$$\tilde{L}^{(2)} \oplus L^{(2)} = \mathrm{mulhi}_M(\tilde{L}^{(1)}) \oplus \mathrm{mulhi}_M(L^{(1)}) = \mathrm{mulhi}_M(L^{(1)} \oplus 1) \oplus \mathrm{mulhi}_M(L^{(1)})\,.$$

Consider an integer X with $Q = \mathrm{mulhi}_M(X)$ and $R = \mathrm{mullo}_M(X)$. Accordingly, $X \times M = Q \times 2^{\mathrm{W}} + R$. A standard ± 1 carry/borrow analysis yields the following:

$$(X \oplus 1) \times M = \begin{cases} (X+1) \times M = Q \times 2^{\mathrm{W}} + R + M & \text{if } X \text{ is even,} \\ (X-1) \times M = Q \times 2^{\mathrm{W}} + R - M & \text{if } X \text{ is odd.} \end{cases}$$

Then, we obtain

$$\mathrm{mulhi}_M(X \oplus 1) = \begin{cases} Q + \lfloor (R+M)/2^{\mathrm{W}} \rfloor & \text{if } X \text{ is even,} \\ Q + \lfloor (R-M)/2^{\mathrm{W}} \rfloor & \text{if } X \text{ is odd,} \end{cases}$$

and more precisely,

$$\mathrm{mulhi}_M(X \oplus 1) = \begin{cases} Q+1 & \text{if } X \text{ is even and } R \geq 2^{\mathrm{W}} - M\,, \\ Q & \text{if } X \text{ is even and } R < 2^{\mathrm{W}} - M\,, \\ Q & \text{if } X \text{ is odd and } R \geq M\,, \\ Q-1 & \text{if } X \text{ is odd and } R < M\,. \end{cases}$$

Thus, with probability $p_{\mathrm{carry}} = (M-1)/2^{\mathrm{W}}$, we have $\mathrm{mulhi}_M(X \oplus 1) \neq \mathrm{mulhi}_M(X)$ (differing by ± 1), and with probability $1 - (M-1)/2^{\mathrm{W}}$, the values are equal.

Next, consider the XOR difference between $\mathrm{mulhi}_M(X)$ and $\mathrm{mulhi}_M(X \oplus 1)$ in the case they are unequal. If Q is a random W-bit word, then we can write:

$$Q \oplus (Q+1) = 2^{\tau_1(Q)+1} - 1\,, \qquad Q \oplus (Q-1) = 2^{\tau_0(Q)+1} - 1\,,$$

where $\tau_1(Q)$ (resp. $\tau_0(Q)$) denotes the number of consecutive least significant ones (resp. zeros) in the binary representation of Q. Assuming that $Q = \mathrm{mulhi}_M(X)$ is uniformly distributed over all W-bit integers, we obtain

$$\tilde{L}^{(2)} \oplus L^{(2)} = \begin{cases} 0, & \text{with prob. } 1 - p_{\mathrm{carry}}\,, \\ 2^d - 1, & \text{with prob. } p_{\mathrm{carry}} \cdot 2^{-d} \text{ for each } d \text{ with } (d \geq 1)\,. \end{cases}$$

Moreover, for the third round, we have

$$\begin{aligned} \tilde{L}_0^{(3)} \oplus L_0^{(3)} &= \left(\mathrm{mulhi}_M(\tilde{L}^{(2)}) \oplus \mathrm{mulhi}_M(L^{(2)})\right)_0 \oplus \tilde{R}_0^{(2)} \oplus R_0^{(2)} \\ &= \left(\mathrm{mulhi}_M(L^{(2)}) \oplus \mathrm{mulhi}_M(\tilde{L}^{(2)})\right)_0 \oplus 1\,. \end{aligned}$$

Furthermore, in the fourth round, due to parity preservation by mullo_M, we obtain

$$\tilde{R}_0^{(4)} \oplus R_0^{(4)} = \tilde{L}_0^{(3)} \oplus L_0^{(3)}\,.$$

Therefore, we can evaluate the matching probability as follows:

$$\begin{aligned}\Pr\left[R_0^{(4)} = \tilde{R}_0^{(4)}\right] &= \Pr\left[L_0^{(3)} = \tilde{L}_0^{(3)}\right] \\ &= \Pr\left[\left(\mathrm{mulhi}_M(L^{(2)}) \oplus \mathrm{mulhi}_M(\tilde{L}^{(2)})\right)_0 = 1\right].\end{aligned}$$

Based on the possible values for $\tilde{L}^{(2)} \oplus L^{(2)}$, it suffices to consider the cases where $\tilde{L}^{(2)} \oplus L^{(2)} = 2^d - 1$. Hence, we obtain the following summation:

$$\begin{aligned}&\Pr\left[R_0^{(4)} \oplus \tilde{R}_0^{(4)}\right] \\ &\qquad = p_{\mathrm{carry}} \cdot \sum_{d=1}^{\mathrm{W}} 2^{-d} \cdot \Pr\left[\left(\mathrm{mulhi}_M(L^{(2)}) \oplus \mathrm{mulhi}_M(L^{(2)} \oplus (2^d - 1))\right)_0 = 1\right].\end{aligned}$$

For the case $d = 1$, it can be shown that

$$\Pr\left[\left(\mathrm{mulhi}_M(L^{(2)}) \oplus \mathrm{mulhi}_M(L^{(2)} \oplus 1)\right)_0 = 1\right] = \frac{M-1}{2^{\mathrm{W}}}.$$

For cases with $d > 1$, we compute these probabilities using the recommended value of the multiplier in Philox–2×64. In this way, we obtain

$$\Pr\left[R_0^{(4)} = \tilde{R}_0^{(4)}\right] \approx 0.56.$$

Finally, by applying the same approach as in the proof of Distinguisher 3, the above 4-round differential-linear distinguisher can be extended to the 5-round distinguisher mentioned in the statement. □ □

In the following, we introduce (multiplicative) differential-linear distinguishers on Philox–4×W that conceptually inherit their core principles from the Philox–2×W variants. Precisely, we introduce non-zero differences into only two of the four branches, either $(Y^{(0)}, Z^{(0)})$ or symmetrically, $(X^{(0)}, T^{(0)})$, while keeping the differences in the remaining two branches at zero. These configurations make the distinguishers for Philox–4×W to be significantly stronger than their Philox–2×W counterparts, yielding effective distinguishers that extend up to 6 rounds. Since the propagation characteristics of the first two configurations are highly analogous to the two-word case, we relegate their detailed statements and proofs to Appendix B. Moreover, by analyzing the propagation of multiplicative differentials more deeply, we find that Distinguisher 5 holds deterministically with probability 1.

Distinguisher 5. *Let* $P = (X^{(0)}, Y^{(0)}, Z^{(0)}, T^{(0)})$ *and* $\tilde{P} = (\tilde{X}^{(0)}, \tilde{Y}^{(0)}, \tilde{Z}^{(0)}, \tilde{T}^{(0)})$ *be two plaintexts for* Philox–4×W *such that* $Y^{(0)}$ *is odd, and*

$$\begin{aligned}\tilde{X}^{(0)} &= X^{(0)} \oplus (2^{\mathrm{W}} - 2), \quad \tilde{Y}^{(0)} = Y^{(0)}, \quad \tilde{Z}^{(0)} = Z^{(0)}, \\ \tilde{T}^{(0)} &= T^{(0)} \oplus \mathrm{mulhi}_{M_0}(X^{(0)}) \oplus \left(M_1 - 1 - \mathrm{mulhi}_{M_0}(X^{(0)})\right), \\ \Big(\textit{resp. } \tilde{Z}^{(0)} &= Z^{(0)} \oplus (2^{\mathrm{W}} - 2), \quad \tilde{T}^{(0)} = T^{(0)}, \quad \tilde{X}^{(0)} = X^{(0)}, \\ \tilde{Y}^{(0)} &= Y^{(0)} \oplus \mathrm{mulhi}_{M_1}(Z^{(0)}) \oplus \left(M_0 - 1 - \mathrm{mulhi}_{M_1}(Z^{(0)})\right)\Big)\end{aligned}$$

Then the corresponding states after six rounds satisfy

$$\begin{aligned}\Pr\left[\left(F_{M_0}(T^{(6)}) \oplus Z^{(6)}\right)_0 = \left(F_{M_0}(\tilde{T}^{(6)}) \oplus \tilde{Z}^{(6)}\right)_0\right] &= 1, \\ (\textit{resp. } \Pr\left[\left(F_{M_1}(Y^{(6)}) \oplus X^{(6)}\right)_0 = \left(F_{M_1}(\tilde{Y}^{(6)}) \oplus \tilde{X}^{(6)}\right)_0\right] &= 1).\end{aligned}$$

Next, we introduce an alternative distinguisher that holds with a lower probability than Distinguisher 5. Its primary significance lies in its input differential pattern, which makes it exceptionally well-suited for counter-based random number generators.

Distinguisher 6. *Let* $P = (X^{(0)}, Y^{(0)}, Z^{(0)}, T^{(0)})$ *and* $\tilde{P} = (\tilde{X}^{(0)}, \tilde{Y}^{(0)}, \tilde{Z}^{(0)}, \tilde{T}^{(0)})$ *be two plaintexts for* PHILOX–4×W *such that*

$$\tilde{X}^{(0)} = X^{(0)}, \quad \tilde{Y}^{(0)} = Y^{(0)}, \quad \tilde{Z}^{(0)} = Z^{(0)}, \text{ and } \tilde{T}^{(0)} = T^{(0)} \oplus 1$$
$$\big(\textit{resp. } \tilde{X}^{(0)} = X^{(0)}, \quad \tilde{Y}^{(0)} = Y^{(0)} \oplus 1, \quad \tilde{Z}^{(0)} = Z^{(0)}, \text{ and } \tilde{T}^{(0)} = T^{(0)}\big).$$

Then the corresponding states after six rounds satisfy

$$\Pr\left[\left(F_{M_0}(T^{(6)}) \oplus Z^{(6)}\right)_0 = \left(F_{M_0}(\tilde{T}^{(6)}) \oplus \tilde{Z}^{(6)}\right)_0\right] \approx 0.56\,,$$
$$\left(\textit{resp. } \Pr\left[\left(F_{M_1}(Y^{(6)}) \oplus X^{(6)}\right)_0 = \left(F_{M_1}(\tilde{Y}^{(6)}) \oplus \tilde{X}^{(6)}\right)_0\right] \approx 0.56\right).$$

For the plaintext pairs in Distinguisher 6, the inputs differ exclusively in the LSB of the rightmost branch when activating the $T^{(0)}$ branch. Consequently, if the LSB of this branch in the first plaintext is zero, the corresponding paired plaintext naturally occurs exactly one counter increment later. This unique property enables the efficient collection of suitable pairs from standard generator outputs in practice, ultimately leading to a ciphertext-only attack.

C Goldreich-Levin in Brief

We recall succinctly how the Goldreich-Levin (GL) procedure recovers the output mask Γ_o that maximizes the *per-key* linear correlation on the last segment E_ℓ, given a fixed input mask Γ_m. Consider the Boolean oracle

$$f_K(Y) = \langle \Gamma_m,\, E^{-1}_{\ell,K}(Y)\rangle \in \mathbb{F}_2.$$

For any mask $A \in \mathbb{F}_2^{64}$, its Fourier-Walsh coefficient (for key K) is

$$\widehat{f_K}(A) = \mathbb{E}_Y\left[(-1)^{f_K(Y) \oplus \langle A, Y\rangle}\right].$$

A direct substitution shows that $\widehat{f_K}(\Gamma_o)$ equals the *per-key* segment correlation q_K of the linear approximation on E_ℓ. The GL task is precisely to recover a mask with a *heavy* coefficient, i.e., $|\widehat{f_K}(A)|$ noticeably above noise.

Algorithmic Sketch (hidden parity with a noise on a 64-bit segment):

- Choose parameters t (subspace dimension), R (iterations), and a detection threshold $\tau > 0$.
- Repeat the following steps R times:
  - Pick a random base point Y_0 and a random t-*coordinate mask* of Hamming weight t, defining an affine sub-cube $Y_0 \oplus V$, $V \in \mathbb{F}_2^t$.
  - Evaluate $g(V) = (-1)^{f_K(Y_0 \oplus V)}$ for all 2^t points, and run a fast Walsh-Hadamard transform (FWHT) to obtain the local spectrum $\widehat{g}(\alpha)$.
  - Record the indices α with large magnitude, e.g., $|\widehat{g}(\alpha)| \geq \tau \cdot 2^t$ (for the unnormalized FWHT). Each such α is the projection of some global mask A onto the chosen coordinates; convert α into per-bit *votes* for the corresponding bits of A.

Table 7: Test vectors for POLSTER. All values are in hexadecimal.

Key	Plaintext	Ciphertext
00000000000000000000000000000000	0000000000000000	305d563c1777c109
ffffffffffffffffffffffffffffffff	ffffffffffffffff	e0b749b865e765dc
00112233445566778899aabbccddeeff	0123456789abcdef	02b1804e5001ae27

- Aggregate votes over the R iterations by per-bit majority to obtain a global candidate mask A; optionally keep the top k candidates.
- Verify each candidate by Monte Carlo: estimate $\operatorname{Cor}_Y\big(f_K(Y) \oplus \langle A, Y\rangle\big)$ from fresh samples and across many keys; accept if $|\mathrm{Cor}| \geq \tau$.

Parameter Guidance (choose by signal strength, not block size):

- t (Subspace Dimension): each GL round uses 2^t oracle calls. As a general rule, pick t around or somewhat above $\log_2(1/|q|)$; e.g., if $|q| \approx 2^{-13}$, then $t \approx 16$ is a good starting point.
- R (Iterations): number of random subspaces. Values in the range 8–64 typically suffice for stable per-bit majorities.
- τ (Detection Threshold): report local coefficients only if $|\widehat{g}(\alpha)| \gtrsim \tau \cdot 2^t$ (unnormalized). Smaller τ detects weaker signals but requires more verification samples.
- Verification Budget: To confirm $|q| \geq \tau$ with small error, use on the order of $M \approx c/\tau^2$ samples per key (constant c depends on the desired confidence).

Complexity and Practice. GL uses $O(R \cdot 2^t)$ oracle calls and $O(R \cdot t \cdot 2^t)$ time (one FWHT of length 2^t per subspace), versus $\Theta(2^{64})$ for exhaustive search on masks. In our implementation we instantiate f_K with the exact inverse $E_{\ell,K}^{-1}$ and re-estimate the final candidates across many keys to suppress key-dependent artifacts. When no heavy coefficient passes the threshold, we fall back to single-bit masks and select by direct correlation estimation.

D Test vectors for Polster

E Usage of Generative AI.

During the preparation of this article, Google's *Gemini 3.1* and OpenAI's *ChatGPT* (GPT-5) were used as tools to support the research and writing process. More precisely, those were employed for text editing, language refinement, and implementing an attack code from its description. All conceptual development, verification of results, and final revisions were conducted by the authors, who take full responsibility of the content.

F AlphaEvolve Prompts and Evaluation

In this appendix, we detail the configuration and experiment setup we used for the LLM-guided evolutionary search. The overall architecture of our setup is laid out in Figure 4. The search agent here used Gemini 3.1 Pro and Flash models, with the LLM model being given the context from Listing F.1.

The core of the evolutionary search is the evaluation function (see Listing F.3), which acts as the fitness function for the search. Each program generated by the LLM is evaluated using this function, specifically it experimentally verifies the probability of the distinguisher over 200.000 samples. This is done by generating a fixed key, creating pairs which have the difference pattern found by the program under evaluation and applying the output mask to calculate the bias. To prevent the model from exploiting trivial solutions, we had to add various checks to force no trivial differences or output masks and score such solutions very low.

For the initial program (see Listing F.2), we just set a single bit difference and output mask. From different experiments we run, this had little influence on how quickly a solution was found.

Listing F.1: Problem Context

```
Your goal is to improve a Python function, `find_best_distinguisher`, that attempts to discover powerful
differential-linear distinguishers for the 7-round PHILOX-4x32 PRNG. The objective is to maximize the
correlation bias of the distinguisher over 7 rounds.

The function you are modifying proposes an input difference function (`diff_func`) and a linear output mask
(`out_mask`). You are highly encouraged to write new, mathematically clever `diff_func` implementations
directly within the Evolve block and test them with specific output masks.

Function Signature:
find_best_distinguisher(time_limit_seconds: float) -> dict

The returned dictionary MUST contain:
1. 'diff_func': A callable that takes (X, Y, Z, T) and returns a modified 4-tuple (X_tilde, Y_tilde, Z_tilde, T_tilde).
2. 'out_mask': A 4-tuple of 32-bit integers (mask_X, mask_Y, mask_Z, mask_T) used to extract the linear bias at the output
      .
3. 'math_repr': A compact string representing the mathematical operations happening for the difference.

Constraints (Enforced by the evaluate function):
1. No Trivial Differences: If your `diff_func` returns the exact same plaintext as the input (X, Y, Z, T)
for more than 5% of the samples, it is considered a trivial difference and will receive a penalty score of
-10000.
2. No All-Zero Mask: The `out_mask` cannot be entirely composed of zeros `(0, 0, 0, 0)`.

Scoring:
The evaluate function generates thousands of random plaintexts, applies your `diff_func` to create pairs,
encrypts them using 7-round PHILOX-4x32, applies your `out_mask` to the ciphertext differences, and
calculates the absolute correlation bias. AlphaEvolve will try to maximize this correlation. A trusted
`get_score(diff_func, out_mask)` function is provided globally for you to test your ideas inside the search
loop.
```

Listing F.2: Initial Program

```
import time, random

def find_best_distinguisher(time_limit_seconds: float) -> dict:
    """
    Search for PHILOX-4x32 distinguishers using differential-linear linearization
    and perfect neutralization of Feistel channels.
    """
    import time
    start_time = time.time()
    M0, M1, mask = 0xD2511F53, 0xCD9E8D57, 0xFFFFFFFF

    # Start with simple single bit input difference and output mask
    best_sol = {'diff_func': lambda X,Y,Z,T: (X^0x80000000, Y, Z, T), 'out_mask': (0, 0, 0, 1), 'math_repr': "init"}

    return best_sol
```

Listing F.3: Evaluation Function

```
from typing import Any, Dict
import random

# --- Ground Truth Validation and Scoring Functions ---

M0 = 0xD2511F53
M1 = 0xCD9E8D57

def _mullo(a, b):
    return (a * b) & 0xFFFFFFFF

def _mulhi(a, b):
    return ((a * b) >> 32) & 0xFFFFFFFF

def _philox4x32_round(X, Y, Z, T, k0, k1):
    X_out = (_mulhi(Z, M1) ^ Y ^ k0) & 0xFFFFFFFF
    Y_out = _mullo(Z, M1)
    Z_out = (_mulhi(X, M0) ^ T ^ k1) & 0xFFFFFFFF
    T_out = _mullo(X, M0)
    return X_out, Y_out, Z_out, T_out

```

```python
def _philox4x32_7_rounds(X, Y, Z, T, key):
    k0, k1 = key[0], key[1]

    C0 = 0x9E3779B9
    C1 = 0xBB67AE85

    for r in range(7):
        X, Y, Z, T = _philox4x32_round(X, Y, Z, T, k0, k1)
        k0 = (k0 + C0) & 0xFFFFFFFF
        k1 = (k1 + C1) & 0xFFFFFFFF

    return X, Y, Z, T

def get_score(diff_func, out_mask, n_samples):
    """
    Evaluates the proposed input difference function against the specified output mask
    over 7 rounds of PHILOX-4x32. Returns the absolute correlation bias.
    """
    if not isinstance(out_mask, (tuple, list)) or len(out_mask) != 4:
        return -10000.0

    mX, mY, mZ, mT = out_mask

    # Check for all-zero mask
    if mX == 0 and mY == 0 and mZ == 0 and mT == 0:
        return -10000.0

    trivial_count = 0
    matches = 0

    # We use a fixed key for the distinguisher evaluation to simulate a known-key / standard PRNG setting
    key = (random.getrandbits(32), random.getrandbits(32))

    for _ in range(n_samples):
        # Generate random plaintext
        X, Y, Z, T = (random.getrandbits(32) for _ in range(4))

        try:
            X_tilde, Y_tilde, Z_tilde, T_tilde = diff_func(X, Y, Z, T)
        except Exception:
            return -10000.0 # Bad function

        # Enforce 32-bit limits
        X_tilde &= 0xFFFFFFFF
        Y_tilde &= 0xFFFFFFFF
        Z_tilde &= 0xFFFFFFFF
        T_tilde &= 0xFFFFFFFF

        if (X, Y, Z, T) == (X_tilde, Y_tilde, Z_tilde, T_tilde):
            trivial_count += 1

        # Encrypt P and P_tilde
        C_X, C_Y, C_Z, C_T = _philox4x32_7_rounds(X, Y, Z, T, key)
        Ct_X, Ct_Y, Ct_Z, Ct_T = _philox4x32_7_rounds(X_tilde, Y_tilde, Z_tilde, T_tilde, key)

        # Compute direct parity of the masked output difference
        dot = ((C_X ^ Ct_X) & mX) ^ ((C_Y ^ Ct_Y) & mY) ^ ((C_Z ^ Ct_Z) & mZ) ^ ((C_T ^ Ct_T) & mT)

        # Fast bit count to compute parity
        parity = bin(dot).count('1') & 1
        if parity == 0:
            matches += 1

    # Penalize if it generates trivial differences too often (>5%)
    if trivial_count > 0.05 * n_samples:
        return -10000.0

    final_corr = abs(2.0 * matches / n_samples - 1.0)
    return final_corr

def evaluate(data: Dict[str, Any]) -> tuple[Dict[str, float], Dict[str, Any]]:
    time_limit_seconds = 45.0

    try:
        proposed = find_best_distinguisher(time_limit_seconds)
    except Exception:
        return {'correlation': -10000.0}, {'image': ""}

    if not isinstance(proposed, dict) or 'diff_func' not in proposed or 'out_mask' not in proposed:
        return {'correlation': -10000.0}, {'image': ""}

    # Final evaluation with larger sample size
    score = get_score(proposed['diff_func'], proposed['out_mask'], n_samples=200000)

    final_solution = {
        'diff_name': proposed.get('diff_name', 'custom'),
        'math_repr': proposed.get('math_repr', 'No mathematical representation provided'),
        'out_mask': proposed['out_mask'],
        'score': score
    }

    outputs = {'best_distinguisher': final_solution}

    return {'correlation': score}, outputs
```